NAVAL POSTGRADUATE SCHOOL
MONTEREY, CALIFORNIA

THESIS

RE-TARGETING THE GRAZE PERFORMANCE DEBUGGING TOOL FOR JAVA THREADS AND ANALYZING THE RE-TARGETING TO AUTOMATICALLY PARALLELIZED (FORTRAN) CODE

by

Pedro T.H. Tsai

March 2000

Thesis Advisor: Debra Hensgen
Second Reader: Rudy Darken

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE March 2000	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE: RE-TARGETING THE GRAZE PERFORMANCE DEBUGGING TOOL FOR JAVA THREADS AND ANALYZING THE RE-TARGETING TO AUTOMATICALLY PARALLELIZED (FORTRAN) CODE	5. FUNDING NUMBERS
6. AUTHOR(S) Tsai, Pedro, T.H.	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT: Approved for public release; distribution is unlimited.	12b. DISTRIBUTION CODE:

13. ABSTRACT *(maximum 200 words)*

This research focuses on the design of a language-independent concept, Glimpse, for performance debugging of multi-threaded programs. This research extends previous work on Graze, a tool designed and implemented for performance debugging of C++ programs. Not only is Glimpse easily portable among different programming languages, (i) it is useful in many different paradigms ranging from few long-lived threads to many short-lived threads; and (ii) it generalizes the concept of intervals over Graze's original definition. Glimpse's portability has been validated by demonstrating its usefulness in performance debugging of both Java programs as well as automatically parallelized FORTRAN programs.

14. SUBJECT TERMS Performance Debugging, Java Threads, Automatically Parallelized FORTRAN applications	15. NUMBER OF PAGES 134
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited

RE-TARGETING THE GRAZE PERFORMANCE DEBUGGING TOOL FOR JAVA THREADS AND ANALYZING THE RE-TARGETING TO AUTOMATICALLY PARALLELIZED (FORTRAN) CODE

Pedro T.H. Tsai
Naval Research Lab, Monterey CA

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 2000

Author: Pedro T.H. Tsai

Pedro T.H. Tsai

Approved by: Debra O. Hensgen

Debra Hensgen, Thesis Advisor

Rudolph P. Darken

Rudy Darken, Second Reader

Dan C Boger

Dan Boger, Chairman
Department of Computer Science

ABSTRACT

This research focuses on the design of a language-independent concept, Glimpse, for performance debugging of multi-threaded programs. This research extends previous work on Graze, a tool designed and implemented for performance debugging of C++ programs. Not only is Glimpse easily portable among different programming languages, (i) it is useful in many different paradigms ranging from few long-lived threads to many short-lived threads; and (ii) it generalizes the concept of intervals over Graze's original definition. Glimpse's portability has been validated by demonstrating its usefulness in performance debugging of both Java programs as well as automatically parallelized FORTRAN programs.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I like to thank Debra Hensgen, this thesis would not have been possible without her guidance and teaching. To my parents, for their support and understanding. Finally, to Paul, who was my brother, mentor, and my best friend, with all my love.

I. INTRODUCTION

This thesis focuses on the design and implementation of a language independent performance debugging suite called Glimpse. Glimpse collects profiling data from the execution of multi-threaded programs and provides visualization tools to help programmers analyze the collected data. In particular, it builds upon the approach taken by Graze [Ref. 1], a performance debugging tool that monitors C++ programs that use the Solaris thread library. Glimpse generalizes Graze, making it both language independent and more functional. A language independent tool is needed to permit monitoring of many threaded programs important to the Navy as well as to all of DoD. Examples of such programs are the Master Environmental Library (**MEL**), which is written in Java; and the US Navy Numerical Weather Prediction (**NWP**) applications, which are written in *FORTRAN*. **MEL** is a new system being developed by Naval Research Laboratory (NRL) for the purpose of disseminating environmental data over the network. It is important in such a system to dynamically identify performance bottlenecks. NWP applications, on the other hand, are legacy *FORTRAN* code that is being ported to high-end workstations (SGI, DEC/Alpha) and automatically parallized. Here bottlenecks may arise simply from executing the code on a new platform and, to make the job more difficult, the people porting the code are not, in many cases, the original programmers.

1. Motivation

The NWP applications, for example, the Navy Operational Global Atmospheric Prediction System (**NOGAPS**) [Ref. 2] and the Coupled Ocean Atmosphere Mesoscale Prediction System (**COAMPS**) [Ref. 3], predict atmospheric parameters such as winds, temperature, pressure, and precipitation. These predictions are based on the solution of sets of thermodynamic and fluid dynamic equations. By integrating these equations forward in time, NOGAPS and COAMPS are able to predict the state of the atmosphere in the near future. Because NOGAPS and COAMPS are compu-

tationally intensive, and the information they produce is perishable, minimizing the execution time of these codes is paramount. Most NWP applications use some form of parallel computing to improve their performance. Rather than reimplementing these applications in a parallel programming language, programmers often only add compiler directives to legacy code. These directives cause the compiler to automatically parallelize the code. Programmers place these compiler directives in what appears to be the comment sections of the source code, usually near the computationally intensive loop constructs. These directives indicate to the compiler that a particular loop may be safely parallelized. The compiler will then perform data flow analysis and attempt to generate the necessary machine code for the parallel execution of the loop.

MEL [Ref. 4], another example of an application that can benefit from performance monitoring, is an online digital library for environmental data and other resources. Environmental resources are often (i) difficult to locate, (ii) frequently duplicative, (iii) independently defined and formatted, and (iv) accessible only through an interface that is unique to each repository. The objective of MEL is to provide the user with a single interface that they can use to discover, query, retrieve, and order environmental data. Conceptually, the MEL system is analogous to the card catalog in a library. The card catalog enables the user to search the entire holdings of the library by specific criteria such as topics, authors and titles. Using the emerging standards for describing geo-spatial data and contents, MEL provides a digital metadata database for environmental resources. Since it is not practical to replicate the existing worldwide system of distributed repositories by creating a single, massive resource site for environmental information, the designers of MEL instead choose a three-tier client-server architecture. The first tier is the user interface to the MEL system. It consists of customers and standard WEB browsers. The second tier is the MEL access site, which consists of both hardware and software. The MEL access site fulfills the dual role of handling incoming queries and orders from the customers,

as well as matching the queries and orders to MEL software resident on the third tier. The third tier is composed of actual resource site databases, including their extraction and delivery processes. MEL software is installed at each resource site to facilitate interactions between the MEL access site and the resource sites. The MEL Service Architecture (**MSA**) is the software library (API's) developed to implement the multi-threaded servers running on each MEL access site and MEL clients running inside the WEB browser. MSA uses the Common Object Request Broker Architecture (CORBA) framework and communicates over the network via the Internet Inter ORB Protocol (IIOP). Because there are typically many users and only a few MEL access sites, the access site server that handles the user query and order is potentially a bottleneck. A performance bottleneck could also occur between the MEL access site and the resource site servers that process the requests.

MEL is an object-oriented client-server application, where as NOGAPS and COAMPS are scientific *FORTRAN* codes requiring substantial floating-point computation. Although the programming language used to implement MEL is quite different from NOGAPS and COAMPS, they share a common feature in that they both use a threaded programming paradigm. Unlike a sequential program, performance bugs in a multi-threaded program can be difficult to find using the conventional performance analysis tools. Particularly difficult to determine are performance bugs due to thread synchronization and communication. For example, in the automatically parallelized *FORTRAN* code, a computational loop might be distributed across several threads, with each thread working on a portion of the loop. Typically, there is an implicit '*barrier synchronization*' after the end of the parallel loop; a thread that completes its portion of work will wait until all other threads have finished their portion of work. If there is a load inbalance between threads, it is possible that some threads will idle for a long time while waiting for threads that are still working. Another type of performance bug is due to the communication overhead between threads. An idle thread might periodically query other threads to see whether they have completed.

Such activity, if performed frequently, adds significant overhead to the execution of parallel programs.

2. Background

Conventional performance debugging tools (such as *prof* and *pixie*) are designed to collect profiling information on sequential code. *Pixie*, an object instrumentation tool, adds profiling code to the executable. The modified executable is then run to generate information on code execution frequency. On the Silicon Graphics systems, *Speedshop* software (an integrated front end to *pixie* and *prof*) reads an executable, partitions it into basic blocks, and writes out an equivalent executable program containing additional code that counts the execution of each basic block. (A basic block is a region of the program that can be entered only at the beginning and exited only at the end). Data collected by *pixie* is then analyzed by *prof* that generates reports on the various statistics such as the frequency of function calls and total percentage of time spent in each function. These conventional tools can provide very useful and detailed information on how a program is spending its time, but they do not provide information on interaction between threads. For example, *pixie* would be able to tell us how much total time a program (all threads) spends in a parallel loop, but it cannot tell us whether some threads are spending too much time idling due to a synchronization barrier.

Parallel debugging tools such as Graze [Ref. 1], Pablo [Ref. 5], PSpec [Ref. 6] and Paradyn [Ref. 7] have attempted to solve these problems. The first three tools are user-controlled while the last, *Paradyn*, searches for types of performance problem which have, in the past, commonly plagued parallel applications. The user-controlled tools allow the user to collect profiling information during the execution of threaded programs and to visualize the information collected so that the user gains insight into the significance of data that was collected in this manner. Furthermore, these tools are more flexible than *pixie* in that they allow users to define which performance data is to be collected. In particular, *Graze*, *PSpec* and *Pablo* allow the user to define

events. An *event* has a name and at least one numeric attribute, the time at which the event occurred. When an event occurs, these systems record substate information corresponding to the event. An intrinsic attribute of an event is the identifier of the thread in which it occurred. This attribute allows the user to distinguish events occurring in different threads. Additionally, in *Graze* and *PSpec*, a user can define an *interval* by specifying a designated start event and an end event. The interval is the concept that allows the user to associate related events. In *Graze*, this association can be further specified by the user who requires that the two associated events have identical attribute values (with the exception, of course, of the thread identifier and time). By allowing the user flexibility in deciding what performance data are to be collected and visualized, experiments focusing on different events and intervals can be conducted.

While these tools are headed in the correct direction, more is needed. In particular, language independent techniques that are also independent of thread paradigm would be useful. Additionally, the interval concept needs to be generalized. Building on the approach taken by *Graze*, this thesis shows that the event and interval concepts can also be applied to both automatically parallelized *FORTRAN* program and to multi-threaded Java programs. A working prototype of such a generalization, *Glimpse*, that resulted from this research, is described.

3. Organization

The rest of this thesis is organized as follows. In Chapter II the background and related work on parallel performance debugging are described. Chapter III focuses on the approach used by *Graze.* Chapter IV discusses the changes needed to generalize *Graze* into *Glimpse.* In Chapter V, the results from testing *Glimpse* with a multi-threaded Java program are presented. In Chapter VI, we describe how the same generalization technique can be applied to automatically parallelized FORTRAN applications. That chapter also explains how *Glimpse* was used to gather and analyze performance data from a computationally intense NWP application, COAMPS. The

final chapter enumerates lessons learned while designing and implementing *Glimpse*, summarizes the contributions of this thesis, and describes suggested future work.

II. RELATED WORKS

In this chapter we describe the approaches others have taken in performance debugging of parallel programs. In particular, we discuss several software tools that are available in this application domain.

In section one, the traditional approaches for measuring performance of programs are discussed. In section two and three, we review two research software tools, *Paradyn* and *Pablo*, that represent more recent developments in the area of parallel program debugging. These tools implement dynamic instrumentation and automatic search for performance bottlenecks, intelligent data reduction, and the idea of adaptive control (self-steering) systems applied to performance optimization. In section four, we describe an integrated performance prediction, measurement, and analysis environment. In section five, commercial software, *TimeScan*, for debugging parallel code is described. We summarize these other approaches in section six.

1. Traditional Approach to Performance Debugging

Performance debugging can be characterized by the following activities: measuring performance data from the application, analyzing the data collected to identify bottlenecks, and optimizing the program code that causes the bottlenecks. In this section, we discuss several approaches that are used for data measurement and analysis, and the trade-off between each approach.

Program Counter Sampling. This approach is also known as *profiling*. It is widely used by the UNIX debugging tools such as `prof` and `gprof`. In this approach, a running process is periodically interrupted by the kernel (or some external process) to record a histogram of the program counter locations. Since each histogram bin can be related to a basic block (function) in the program, an estimate of the total time spent in a particular basic block (function) can be obtained by multiplying the corresponding histogram bin value with the known sampling period. Additional information such as call stack can also be recorded at the sampling point to provide

information about how the program counter gets there. This information allows the post-processing program to compute a duration known as the inclusive time of a **function**, that is the time spent in a function and all other functions that it calls. Without the call stack information, only the exclusion time is known. The sampling rate can be set by the user to control the resolution and the amount of data recorded.

On some systems, instead of using a timer, a hardware performance counter can be used to trigger the program counter sampling. For example, on the SGI R10000 systems, one can request the kernel to examine the program counter when the hardware performance counter specified by the user overflows, and record a histogram of the value of the program counter at overflow. The types of hardware performance counters that can be specified for the sampling purpose are counters that record TLB misses, primary or secondary data cache misses, primary or secondary instruction cache misses, etc.

The data collected by program counter sampling is statistical in nature, varying from run to run. Because the data collection process is external to the program being analyzed, no modification to the source code or the object code of the program is needed. This method has the advantage of low overhead when compared to other data collection methods.

Basic Block Counting. This technique counts the number of times that a basic block (function) in a program is executed. Because the counting is not a statistical measure, the observed frequencies are exact. However, the program needs to be instrumented with code to count the number of times each basic block (function) is executed. The instrumentation is typically performed on the object file. For example, on most UNIX system, an object instrumentation tool such as `pixie` reads the executable file, and writes out an equivalent file containing additional code that counts the number of times each basic block (function) executed. To obtain timing information, this technique must either periodically timestamp the recorded count data, or else use a machine model to compute the instruction cycles executed for each basic

block (function), and then infer the time spent in each basic block (function) from the number of instructions executed. The time estimate obtained using the machine model assumes that instructions are executed in an idealized condition.

Event Tracing. In this approach, a program is modified to include data logging code to record specific events during the execution. To instrument the program for event tracing, one can annotate the source code and then process the modified code through a pre-compiler, which translates the programmer's annotation into actual code. Another approach is to provide a set of event-logging API's and a library. The programmer can then instrument the application by inserting data logging calls at the appropriate locations in the code, and then compile the application code linking with the event-logging library.

The types of events that can be recorded include procedure entry or exit points, read and write function calls, or any other location specified by the programmer. Comparing to the profiling and basic block counting approaches, event tracing generates a complete sequence of events that describes the behavior of the program; thus it is the most general instrumentation approach. Because each event must be timestamped and recorded separately, and additional substate information must be recorded at each occurrence of an event, the potential data volume for the event tracing is large. Statistical techniques have been proposed as a means to reduce the amount of the data recorded while still provides an accurate description of the program behavior [Ref. 8].

2. Paradyn Parallel Performance Tools

Paradyn is a tool developed at University of Wisconsin-Madison for debugging parallel programs [Ref. 7]. *Paradyn* has the following characteristics: (1) it is designed to monitor long running program and large program with thousands of procedures; (2) it uses well-defined data abstractions to describe performance related problems; (3) it provides the ability to automatically search for bottlenecks that are known to affect the performance of parallel program; (4) it uses dynamic instrumen-

tation to instrument only those parts of programs relevant to finding the current performance problem; and (5) to leverage off existing visualization tools, *Paradyne* provides a standard interface to the performance data that allows the user to incorporate external visualization programs for examining the performance data. In the following discussion, we describe the approaches and components used by *Paradyn* to implement the functionality listed above. The components that make up *Paradyn* are the Performance Consultant, the Data Manager, the Metric Manager, the Instrumentation Manager, and the Visualization Manager.

Performance Data Abstractions. Paradyn uses two basic data abstractions for collecting, communicating, analyzing, and presenting performance data [Ref. 7]. The abstractions are the metric- focus grid and the time-histogram. A metric-focus grid consists of two orthogonal lists of information. The first list is a vector of performance metrics such as CPU utilization, memory usage, and counts of floating point operations. The second list, focus, is a specification of a part of a program expressed in terms of program resources. Typical resource types are synchronization objects, source code objects (procedures), threads, and processes. The combination of a list of performance metrics with a list of program resources forms a matrix (called a grid in *Paradyn*) with each metric listed for each program resource. The elements of the matrix can be single values, such as an average, a minimum or maximum value of a metric, or time-histograms.

A time-histogram is an array whose buckets store values of a metric for successive time intervals; *Paradyn* uses time-histograms to store metric values as they vary over time. The user can control the amount of data recorded and the resolution of the data by setting the total number of the buckets and the width of the bucket (i.e., the time interval). If a program runs longer than the bucket width times the number of buckets, *Paradyn* doubles the bucket width and re-bins the previous values. The process of doubling the width of bucket is repeated each time all of the buckets are filled. This re-sampling technique reduces the rate of data collection and allows *Para-*

dyn to monitor long-running programs while maintaining a reasonable representation of a metric's time-varying behavior [Ref. 7].

Automatic search of performance problems. To assist the user in locating performance problems in the program, *Paradyn* uses a well-defined notion, called the W3 Search Model, that organizes information about the types of problems found in programs and the various components contained in the current programs. Performance problems are found by searching through the space defined by the W3 model.

The W3 Search Model abstracts those aspects of a parallel program that can affect the performance into three domains: (1) *Why* the application performance is poor, (2) *Where* the performance problem is, and (3) *When* the problem occur. The "why" axis contains common types of performance problem that occur in parallel programs. These potential performance problems are represented as a set of hypotheses and tests. Each hypothesis can have sub-hypotheses, which narrows down the performance problem to a more specific aspect of the program behavior. For example, one hypothesis might be that a program is spending too much time on synchronization. The synchronization bound problem can be further attributed to two sub-hypotheses: (1) too many synchronization operations, or (2) high synchronization blocking time. By organizing classes of performance problems into a hierarchical order, the W3 search model allows the user to "drill down" to a specific cause of the performance bottleneck.

The "where" axis represents program resources in which performance problems lie. Searching along the where axis pinpoints the problem to a specific program component. Using the previous example, a "why" search might identify that a program is synchronization bound, a subsequent "where" search may isolate one synchronization object from among the many synchronization objects as the primary culprit. In *Paradyn*, the program resources are organized into different type of hierarchies, each resource hierarchy representing a related group of "focuses" that can be measured. For example, to identify which synchronization object is the primary

bottleneck, the search along the "where" axis starts the root of the hierarchy *SyncObject*. The next level contains different types of synchronization objects such as the *Semaphore*, the *Lock*, and the *Barrier*. Below the Lock and Barrier abstraction levels are the individual locks and barriers used by the application. The children of Semaphore are individual semaphores used in the application. Another example of a resource hierarchy is the *Procedure* abstraction, under which lies objects such as main.c, read_socket.c, write_socket.c, etc. Other *Paradyn* resource hierarchies include *Machine* (which contains sub-objects such as CPU 1, CPU 2, etc), *IO*, *Memory*, and *Process*. By abstracting different program resources into a separate hierarchy, *Paradyn* allows the user to concentrate on one abstraction at a time when searching for performance problems.

The third axis of the W3 search model is the "when"-which is used to identify at what time the application runs poorly. Programs have distinct phases of execution and the "when" axis represents periods of time during which different types of performance problems can occur. For example, a program may consist of three phases of execution: initialization, computation, and output. Within a single phase of a program, the performance tends to be uniform. However, when a program enters a new phase, its behaviors might change significantly. As a result, decomposing a program's execution into phases provides a convenient way for programmers to understand the performance of their program. Searching along the "when" axis involves testing the hypotheses for a focus during different intervals of time of the application execution [Ref. 7].

Paradyn's Performance Consultant module can automatically discover performance problems by searching through the space defined by the W3 Search Model. Refinements are made across the "where," "when," and "why" axes without involving the user. The search is conducted by considering a list of possible refinements along each axis, then ordering this list using internally defined hints. The Performance Consultant selects one or more refinements from an ordered list. If the selected refinement

is not true, the next item from the ordered refinement list is then evaluated. Paradyn can conduct a fully automatic search or allow the user to make manual refinements to direct the search.

The Performance Consultant is also responsible for directing the data collection process. It makes requests to the Data Manager and receives performance data from the Data Manager. The data collection process is described next.

Dynamic Data Instrumentation. Paradyn uses dynamic instrumentation to instrument only those parts of the program relevant to finding the current performance problem. Dynamic instrumentation defers instrumentation of the program until it is in execution and then inserts, alters, and deletes instrumentation during program execution.

Requests for dynamic instrumentation are made by the Data Manager in terms of a metric-focus grid. The requests are translated into instructions for insertion into the program. The translation is done in two steps. First, the Metric Manager translates the metric-focus requests into machine independent abstractions. Next, the Instrumentation Manager converts the machine independent representation into machine instructions for inserting into the application.

The machine-independent abstractions are expressed using points, primitives, and predicates. *Points* are locations in the application's code where instrumentation can be inserted (currently, the points understood by Paradyn's data collection facility are procedure entry, procedure exit, and individual call statements.) *Primitives* are operations that change the value of a counter and timer, e.g., set counter, add to counter, subtract from counter, set timer, start timer, and stop timer. Counter and timer are the two types of instrumentation supported by *Parady's* Instrumentation Manager: counter counts the frequency of some event in the application, and timer measures the interval between events. *Predicates* are conditional statements that guard the execution of primitives. They consist of a Boolean expression and an action. The Boolean expression can be computed using counters, parameters to a

procedure, return values from a procedure, or numeric or relational operators.

Paradyn's Instrumentation Manager performs the translation of points, primitives, and predicates into machine-level instrumentation. When Paradyn is initially connected to an application process, the Instrumentation Manager identifies all potential instrumentation points by scanning the application binary's image. Procedure entry and exit, as well as call to procedure are detected and noted as points. After Paradyn is connected to the application, the Instrumentation Manager waits for the requests from the Metric Manager. The requests are then translated into machine code fragments, called *trampolines*, for insertion into the binary imagery of the application process.

Two types of trampolines, base trampolines and mini-trampolines, are used. A base trampoline is inserted as follows. The machine instruction at the instrumentation point is replaced with a branch to the base trampoline, and the replaced instruction is relocated to inside the base trampoline. The base trampoline contains calls to mini-trampolines. The calls to mini-trampolines can occur both before and after the relocated instruction. A mini-trampolines is code that evaluates a specific predicate or executes a single primitive [Ref. 7]. Paradyn's Instrumentation Manager is responsible for generating the appropriate machine instructions for the primitives and predicates requested by the Metric Manager, and then transfering these instructions to the application process via a variation of UNIX `ptrace` facility. [1]

If the Performance Consultant determines from the data collected that a hypothesis is no longer valid, the primitives and predicates associated with testing that hypothesis can then be removed from the application process by the Instrumentation Manager.

Open Interface to the Performance Data. Once the instrumentation has been

[1]ptrace, process trace, is a UNIX system call that allows the parent process to control the execution of a child process. The parent can examine and modify the "core image" of a child process in the stopped state, and then cause the child process to continue. The UNIX debugging tool **dbx** uses ptrace to implement a breakpoint in the user's program.

inserted into the application, the data is sent back to the Data Manager for processing by other Paradyn modules. Paradyn provides a library and remote procedure call interface to access the performance data in real-time. Visualization modules (*visi's*) are external processes that use this library and interface. When a visi requests performance data from Paradyn, that request is sent to the Data Manager. If the request data is already being collected, the Data Manager will send the current values to the visi, and provide continuous updates as additional data are collected. If the requested data is no being collected, the Data Manager will request that the Instrumentation Manager start collecting it. Paradyn currently provides visi's for time-histogram plots, bar charts, and tables. The visi interface and library also can provide performance data for other uses, such as evaluating performance predicates for application steering, or logging performance data for experiments [Ref. 7].

In addition to Paradyn's basic data type, counter and timer values, performance data from external sources can also be collected. For example, some systems provide hardware-based counters that collect statistics on page faults, data cache misses, instruction cache misses, and memory usage activity. Data from these external sources can be integrated into the Paradyn instrumentation, and subjected to the same predicate evaluation as other performance metrics. For example, if a system provides a counter for the cumulative number of page faults in a process, then Paradyn's Data Collection Facility can read this counter before and after a procedure executes to determine the approximate number of page faults occurring in that procedure.

3. Pablo

Pablo was created by a research group at the University of Illinois at Urbana-Champaign. In this section we describe some of the work done by the Pablo group in the area of performance analysis of parallel systems. In particular, we describe these research on closed loop adaptive performance systems, and in the area of intelligent

performance data reduction to minimize the overhead associated with performance instrumentation.

The motivation behind the development of adaptive performance monitoring and application-steering systems comes from the following observations [Ref. 9]:

1) The traditional performance debugging processes are characterized by the following activities: (a) Application code is instrumented automatically by object code modifying programs or by compilers, or manually by inserting calls to the instrumentation library. (b) After instrumentation, performance data are captured from running one or more program executions. (c) The performance data are visualized and analyzed by the programmer to identify the bottlenecks. (d) Finally, based on measurement and analysis, either the program code that causes the bottleneck is modified or the runtime system policies are adjusted to better match the program resource requests.

2) As parallel computing evolves from homogeneous parallel systems to distributed collections of heterogeneous systems, application tuning and optimization problems become more complex. The time-varying resources of computational environments further exacerbate these problems. Moreover, the performance of parallel application is sensitive to slight changes in the application code, and to continually evolving system software.

3) Although effective for application codes with repeatable behavior, the traditional *post-mortem* tuning model is ill-suited to the parallel application with time varying resource demands that executes in a distributed heterogeneous environment. Not only may the execution context not be repeatable across program executions, resource availability could change during execution.

To address the issues of heterogeneous and dynamic computing environments, the Pablo group has developed a close loop performance analysis and adaptive control system. This system, called the "*Autopilot*" ([Ref. 8], [Ref. 9]), contains the following components:

(1) *Decision procedures* that determine how and when the system should adjust resource allocation policies and system parameters. (2) *Distributed performance sensors* that collect performance data for decision procedures. (3) *Resource policy actuators* that implement changes to the system parameters and policies in response to decisions.

Fuzzy Logic Decision Procedures. The *Autopilot's* decision procedures accept data from distributed sensors as inputs and use actuators to implement the results of decision processes. There are several traditional techniques for implementing such a decision mechanism, including decisions table and trees. A decision table for resource management would typically contain one dimension for each of the key performance sensor values (e.g., file read request sizes and cache hit ratios). Each dimension is then partitioned into a number of operating range (i.e., small, medium, and large read requests), and a policy and its associated parameters would be associated with each table entry. During uses, policies are identified via table lookup using the current sensor values.

Constructing a decision table to optimize performance presumes knowledge of precise mapping between the resource optimization policies and the sensor parameters. Furthermore, as the number of sensor parameters increases, the storage space to fully discretize the sensor space and associate policies can grow rapidly. Consequently, the designers of Autopilot choose to use fuzzy logic to implement decision procedures. The fuzzy logic system allows manipulation of linguistically described concepts through use of common sense knowledge, e.g., file prefetching benefits small, sequential reads.

Performance Sensor. The Autopilot performance instrumentation is based on a set of distributed sensors that extracts information from the execution application, which may be physically distributed over the network. A sensor has a set of associated properties that are defined at the time it is created. These properties typically include sensor name, type, network IP address, and any user-defined attribute-value pairs.

Sensors can collect data in either asynchronous or synchronous mode. In the asynchronous mode, a separate monitoring thread records the values of the program variables of interest at intervals specified by the client or at the time when the sensor is created. In synchronous mode, the sensors are inserted in either the source code or the object code of the application.

To reduce the amount of data collected locally by the sensor, a sensor can apply a data transformation function to raw data before recording them. The *attached functions* can compute simple statistics (e.g., sliding window averages) or more complex transformations. For example, one type of transformation is to generate qualitative file access pattern descriptions from file input/output request measures (e.g., converting a sequence of file seek operations to sequential, strided, or random access description that characterize the file I/O performed by the application).

Two additional services, a naming service and a client service, are provided to facilitate communication between sensors and decision procedures, and between decision procedures and policy actuators. The naming service supports registration of remote sensors and actuators, and it handles property-based requests for sensors and actuators by the remote clients.

The client service of the *Autopilot* is based on the *Nexus* communication layer [Ref. 10]. Nexus creates a global address space that encompasses all processes executing on a network. Before a client can communicate with a sensor or actuator, it must first obtain the *startpoint* and *endpoint* of sensors or actuators. Similarly, sensors and actuators must obtain the *startpoint* and *endpoint* of their clients. The term *startpoint* and *endpoint* refer to an address in the Nexus global address space. Together these capabilities allow decision procedures to acquire and manage remote sensors and actuators without knowledge of their physical location or creation times.

Policy Actuator. Autopilot actuators allow clients to modify the value of application variables and to remotely invoke application-level functions. Typically, actuators are used to change the resource management policy (e.g, changing file caching

policies). Like the sensors, actuators have associated properties such as name, network IP address, and attached functions.

The following example (from [Ref. 11]) illustrates how the sensors, fuzzy logic decision procedures, and actuators might be used to control file prefetching in an adaptive input/output system. A fuzzy logic controller relies on fuzzy sets to represent the semantic properties of each input (sensor) and output (actuator). The input values of the fuzzy variables are then mapped to the output space by a set of IF-THEN rules.

Figure 1 shows the basic flow of information through the fuzzy logic decision mechanism. The *Autopilot* sensors provide a time-varying stream of file read access classifications. The fuzzification step converts sensor inputs to a value (HIGH, MEDIUM, LOW) for the `ReadClassification` fuzzy variable. The following set of simple fuzzy rules are used to determine the value of output fuzzy variable `PrefetchingFactor`:

```
if ReadClassification = SEQUENTIAL then
   PrefetchingFactor  =  HIGH
if ReadClassification = RANDOM     then
   PrefetchingFactor  =  LOW
if ReadClassification = UNKNOWN    then
   PrefetchingFactor  =  MEDIUM
```

After defuzzification, the value of the `PrefetchingFactor` defines the action taken by an *Autopilot* actuator to adjust the number of blocks that are prefetched.

The rule sets used by the decision procedures are architecture independent; neither the source of fuzzy inputs nor the sink of the fuzzy outputs is specified. The value of `ReadClassification` is an abstraction whose value can be bound to a sensor value, or a classification, or even the output of another decision procedure. Similarly, `PrefetchingFactor` is an abstraction of an actuator, with no implicit mapping. Furthermore, one can experiment with different sensors, choose different actuator policies,

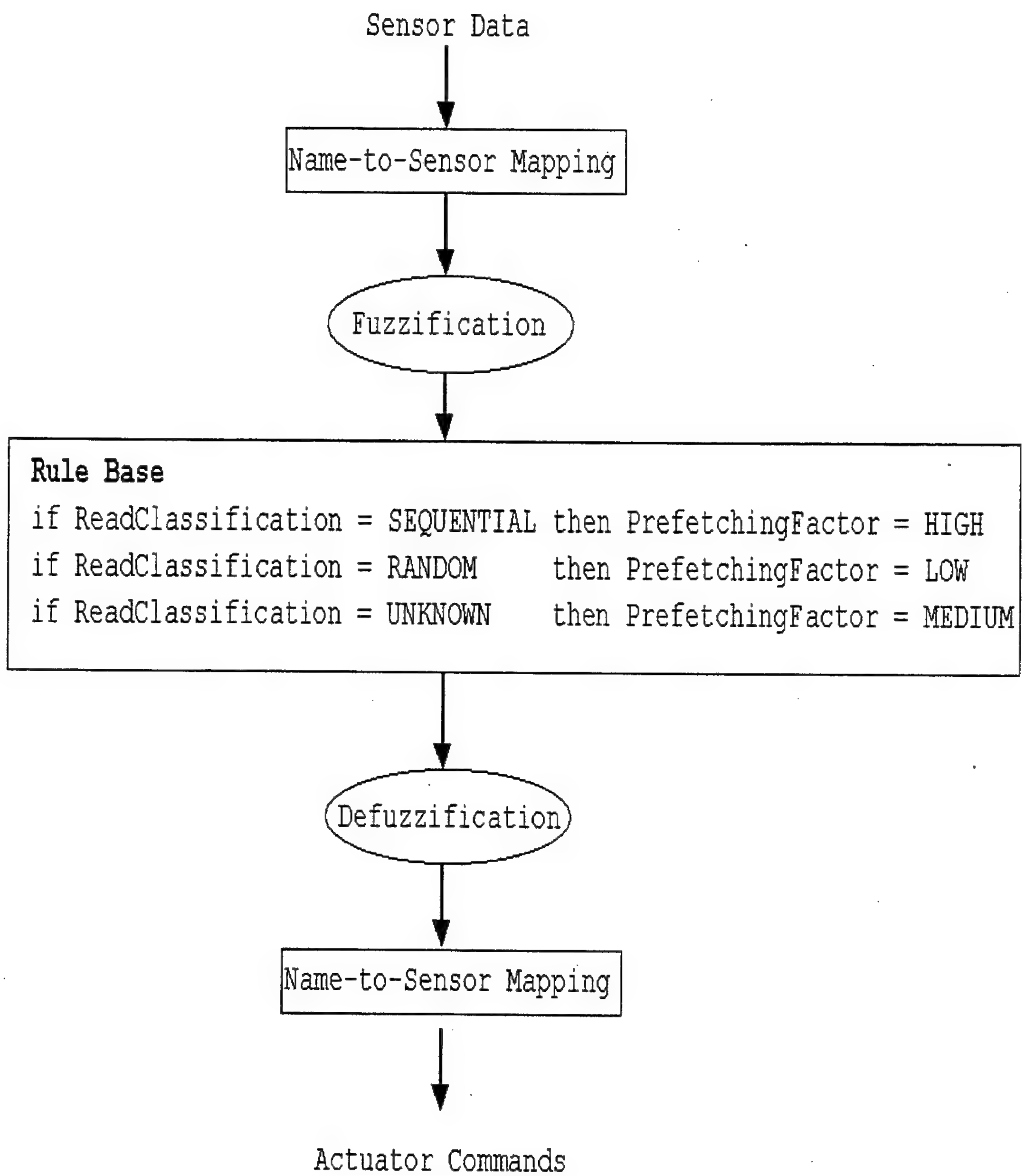

Figure 1. Data flow diagram for the Pablo's `Autopilot` decision mechanism.

or even control different systems by simply binding the inputs and outputs of decision procedures to different sensors and actuators.

4. Delphi

Delphi [Ref. 12] is a new performance environment under development by research groups from University of Illinois, Indiana University, University of Wisconsin, and the Argonne National Lab. Its aim is to provide an integrated performance prediction, measurement, and analysis environment for programmers to evaluate the software and hardware design choices, for both existing and proposed systems. These systems can range from a multi-threaded application running on single processor or multiprocessor machines, to object-oriented application running in a heterogeneous and distributed environment (e.g., CORBA applications). *Delphi* builds on the concepts, experience, and software from several current projects, including the *Pablo* and *Paradyn* performance analysis and measurement tools, the HPC++ and Polaris FORTRAN compiler systems, and the Globus metacomputing system.

An important concept of *Delphi* is the idea of performance prediction. To implement this capability, the *Delphi* framework includes: (1) compilers that emit code annotated with symbolic, execution-cost expressions, and embed calls to instrumentation library in the generated executable; and (2) computation models of key system components, including task schedules, memory, I/O component, network communication. *Delphi's* cost model can produce bounding estimates for the various phases of the program using the compiler- derived data on symbolic program variables and performance measurements from the execution of the instrumented code (the latter provide calibration data and an input-dependent aspect of program execution).

5. TimeScan

The *TimeScan* Event Analysis System is a commercial software tool for debugging, analyzing, and tuning the performance of single or multi-process programs, including programs using light-weight threads. The TimeScan software [Ref. 13] con-

sists of two components: (1) an event logging library (ELOG), and (2) a TimeScan viewer that displays event and state information. The following describes the basic concepts used in the TimeScane software:

event. Events are any program actions, value changes, or procedure calls that a user wants to monitor in order to understand the behavior of the program. For TimeScan, each type of event is identified by an event ID (a unique integer value). All events with the same event ID are interpreted as an *instance* of the same type of event. When events are recorded using ELOG library, ELOG stores a record of data for each event instance. Each logged event record contains a timestamp, an event ID, a data type tag, and a user data item.

event trace. An event trace contains event records from a process (or a thread) in the order in which they occurred. Each event trace is uniquely identified by the hostname, process ID, and thread ID (for program using light-weight threads.)

state. States represent time-spans that are marked by a starting and ending event. For example, by defining a state between a lock-request and lock-granted event, one can determine know how much time a program is waiting for a lock. Currently, only events occurring in the same event trace can be used to form a state.

event log. A log file contains one or more event traces and state information.

To use the TimeScan Event Analysis System, a user would perform the following steps [Ref. 13]:

1) Use the ELOG library functions [Ref. 14] to instrument the program, and compile and link the program with the ELOG library. 2) Run the instrumented program to generate the event log. 3) Run the TimeScan viewer to examine and analyze the event log and state information.

The ELOG library provides functions (for C and C++ programs) to initialize a log file, define event name and ID, define state, log event, and to handle various error conditions. The following pseudo C-code shows how one might instrument a multi-process program with ELOG:

```
#include <stdio.h>
#include ''elog.h''   /* Header file for ELOG function prototype. */

#define START_SEND  1      /* Define event ID */
#define END_SEND    2

int main(int argc, char* argv[])
{
/* Create Event Log. */
ELOG_INIT(''network.elog'');

/* Register Events */
ELOG_DEFINE(START_SEND,''Start writing to socket'',0);
ELOG_DEFINE(END_SEND,''End writing to socket'',0);

/* Define a State called SENDMSG between events START_SEND and
   END_SEND. */
ELOG_DEFINE_STATE(START_SEND, END_SEND,''SENDMSG'');

/* Create 4 child processes to do the work. */
for ( i = 0 ; i < 4; i++ )  {
    status=fork();
    if (status == 0 ) break; /* child process */
}
if (status != 0 ) exit(0);  /* Parent exits. */

/* Allocate a memory buffer for the event trace.  Each thread or
   process has its own memory buffer. */
ELOG_SETUP(''network.elog'',0,0);

/* Log the Starting event. */
ELOG_LOG(START_SEND, 0);

write_to_socket(data);

/* Log the ending event. */
ELOG_LOG(END_SEND, 0);

/* Flush the buffer to event log file. */
ELOG_OUTPUT();
}
```

The call to ELOG_INIT initializes the log file. Next, we defined the START_SEND and END_SEND event, and a state bounded by the two events. After the call to `fork()`, the parent process exits and the four child processes continue to execute. The first thing the child process does is to call ELOG_SETUP. The ELOG_SETUP allocates a memory buffer to store events for that given child process. The first argument to ELOG_SETUP specifies the log filename of the memory buffer to write to. (In this example, a single log file is used to store event trace from the four processes.) The second argument specifies the tag to associate with the memory buffer. A value of 0 causes the use of the default tag, which is derived internally by the ELOG library from the process and thread ID. The third argument specifies the size for this buffer. Using the default value of 0 will allocate space for approximately 1000 events. After the memory buffer is set up, calls to ELOG_LOG add the event instances to the buffer. Finally, ELOG_OUTPUT flushes the content of buffer to the log file.

Once the log file is generated, the user can visualize and analyze the data using the TimeScan viewer. The TimeScan viewer provides a facility to display event records and states as a function of time, with the Y-axis representing the trace from a different process or thread, and X-axis representing the time of the program execution. The viewer also provides support for viewing a subset of events and states, for editing the display symbol and color of the events and states, and for displaying histograms of state durations [Ref. 13].

6. Summary

In this chapter, we provide an overview of some of the existing tools for performance debugging and analysis of parallel programs. In particular, the dynamic instrumentation and automatic search of performance bottlenecks technique implemented by *Paradyn*, and the adaptive control and steering system for performance optimization by *Pablo* are discussed. Recent research efforts (*Delphi: An integrated, Language-Directed Performance Prediction, Measurement and Analysis Environment,*

see [Ref. 12]) aim to combine both dynamic instrumentation and automatica search for performance bottlenecks (W3 search model), with adaptive resource management, compiler integration, and performance prediction capability into an integrated environment.

THIS PAGE INTENTIONALLY LEFT BLANK

III. BACKGROUND ON GRAZE

Graze is a framework for the collection, visualization, and analysis of performance data from applications with multiple threads of control. It was developed at the University of Cincinnati and has been successfully used to identify performance bottlenecks in a multi-threaded VHDL simulation application [Ref. 1]. Unlike the traditional profiling tools such as *prof* and *pixie*, *Graze* lets the user specify exact locations in the program and types of run-time data to be recorded. The data collected by *Graze* are thread specific, that is data can be attributed to a specific thread of the monitored program. The ability to examine user-specified run-time data from the applications and to correlate information from different threads is important for identifying possible performance problems such as the overhead due to communication and synchronization among threads in a program. Such problems are difficult to detect from the conventional profiling data.

The *Graze* framework is comprised of three components: a specification language, a data collection facility, and a generic data visualization facility. In the following sections, we will describe each component of *Graze* in more detail.

1. Graze Specification Language

The *Graze* specification language [Ref. 1] is similar to the performance specification language used by PSpec. PSpec is a system designed for automated performance verification. PSpec uses assertion checking. Its specification language is designed for specifying performance assertions that are checked at run-time [Ref. 15]. For example, the following PSpec specification defines two events, an interval, and an assertion to check the performance of an I/O operation executed by a program:

```
timed event StartRead(); EndRead().
interval Read = s:StartRead, e: EndRead
        metrics time= ts(e) - ts(s)
        end Read.
assert { & r:Read: r.time < 10 ms }.
```

In this example [Ref. 15], StartRead and EndRead are declared to be timed events, which means they have implicit timestamp attributes. A Read interval starts with an event of type StartRead and ends with the next event of type EndRead after the start event. The variables s and e signify the start and end events for an interval of type Read. Each Read interval has a time metric whose value is the difference of its start and end timestamps (ts is a PSpec built-in function that returns the timestamp of an event.) The assert statement checks the elapsed time of any Read operation performed by the program. The statement `assert { & r:Read:  r.time < 10 ms }` can be read as: "for all intervals, r of type Read, the value of r's time metric is at most ten milliseconds."

The *Graze* specification language uses several concepts from the PSpec language; in particular, the event and interval constructs. An event denotes a specific point of interest during the execution of an application; it has a type name and associated attributes. Every *Graze* event has two intrinsic attributes: its owner (a value that identifies the thread that produced the event) and a timestamp. Additional attributes can be specified by the user to cause *Graze* to obtain more detailed run-time information. Using a message passing application as an example, the following entries specify events corresponding to the preparing, sending and receiving of a message [Ref. 1]:

```
event Preparemsg() = diamond;
event Sendmsg(src, seq) = plus;
event Recvmsg(src, seq) = box;
```

The arguments src and seq are user-specified attributes that identify the sender and sequence number of the message sent and received. There are no hard-coded limits on the number of such attributes that an event type may have, however, in the original design of *Graze* these attribute values are restricted to the integer type. The tokens after the '=' are directives to the *Graze* visualization programs; in this case they specify the graphical symbol for depicting the respective event type.

A user can associate two related events by specifying an interval definition. An interval is bounded by a designated start and end event type. For example, the following specifies the interval between `Preparemsg` and `Sendmsg` events:

```
interval Write [ p: Preparemsg -> s:Sendmsg ] = line;
```

In the `Write` interval defined above, the syntax `p:Preparemsg` → `s:Sendmsg` tells *Graze* that after it finds each `Preparemsg` event it should find the next `Sendmsg` in the same thread. Each matching pair of `Preparemsg` and `Sendmsg` events make up an instance of interval `Write`.

In addition to the above interval specification, *Graze* provides another way of specifying the bounding condition between two events. Unlike the previous definition of the `Write` interval, this bounding condition does not require that the start and end events occur in the same thread. Such an interval specification is given below:

```
interval Transit [ s:Sendmsg -> r:Recvmsg ] Match = line;
```

The keyword `Match` tells *Graze* to create instances of interval `Transit` by matching all attribute values of the start and end events, except the timestamp and owner of that event (thread identifier). In this case, the bounding condition is equivalent to `s.src==r.src && s.seq==r.seq`. This type of interval specification is used for describing related events occurring in different threads. For example, in the message passing application, messages are typically transmitted by one thread and received by another thread within the same application. Therefore, using the `Match` interval specification is more appropriate.

Given an interval specification, *Graze* allows for either forward or backward searching of ending events. For example, in the following interval specification:

```
interval BackTransit [ r:Recvmsg <- s:Sendmsg] Match = line;
```

The left arrow symbol tells *Graze* to search backward in time from the point of a `Recvmsg` event instance until a corresponding `Sendmsg` event with matching attribute

values is found. If a matching event is found, then the *Graze* visualization programs marks this pair as an interval. These programs then move forward to the next starting event and repeat the backward search for a matching event, until all intervals meeting the specification are found.

As discussed in the next two sections, in addition to the specification language, *Graze* consists of a Data Collection Facility and a number of visualization toolkits. The specification language described in this section is used by both of these components, although some of the language is ignored by the Data Collection Facility. For example, interval specifications and event shapes are ignored by the Data Collection Facility because they are irrelevant during the run-time of the job being executed.

2. Data Collection Facility

Using the specification provided by the user, the *Graze* framework automatically generates a custom data collection facility for an application [Ref. 1]. A *Graze* utility program `builder` (written using lex and yacc tools) parses the specification and generates a logging function for each event type. The function name is simply the event type name with the suffix "_Stamp" appended to it. If the event type has optional attributes, then the function has formal parameters matching those attributes [Ref. 1]. For example, the C++ function prototypes for the `Sendmsg` and `Recvmsg` events discussed above are:

```
void Sendmsg_Stamp (int src, int seq);
void Recvmsg_Stamp (int src, int seq);
```

The actual code created by the `builder` program is system- and thread-library dependent. The original version of *Graze* only generates logging functions for multi-threaded program that use SUN Solaris thread library. The logging function writes event data to a log file. A logging function, when invoked, performs the following actions: (1) determines the thread from which it is being called; (2) finds the open file descriptor for the log file associated with the calling thread; (3) invokes a system-dependent timing function to obtain a timestamp; (4) writes the event type, timestamp and actual

parameter values to the output file. To minimize the overhead associated with the data logging, *Graze* implements an internal buffer for caching the event data. When the buffer is full, the data is written to the output file. This technique minimizes the number of writes performed.

To instrument an application, the programmer annotates the code using event notation. An example of event notation is:

```
Sendmsg(self, SequenceNumber);
```

A *Graze* specification language preprocessor should replace such lines with the call to the corresponding function, which, in this case is: [1]

```
void Sendmsg_Stamp(self, SequenceNumber);
```

After the application is executed, a set of log files is produced. Each log file contains information about events collected from a single thread in the instrumented program. The filename of the log file contains an integer that identifies the thread associated with the given event stream.

The structure of the log file consists of a header record, followed by a stream of event records. The integer value -1 is used as a sentinel to indicate the end of the log file. The header record consists of a single precision (4 byte) integer (0x12345670) and a double precision (8 byte) integer (0x123456789abcdef0). *Graze* visualization programs (`nibble` and `gorge`) read the header record to verify that input files are valid *Graze* log files and to determine the byte ordering (endianess) of the data. Each event record consists of an event type identifier, timestamp, and a list of integer values corresponding to the user specified event attributes, if any. All values in the log files are either single of double precision integers. The following layout shows the structure of the log file:

[1] Currently, the programmer must insert calls to the event logging functions, but it would be easy to modify the front-end of GNU compilers to perform this replacement.

```
header record:
                integer (4 byte)
                double precision integer (8 byte)

event records:
                event type identifier (integer: 4 byte)
                time stamp (double precision integer: 8 byte)
                optional program state information
                { integer, integer, ... }
                event type identifier (integer: 4 byte)
                time stamp (double precision integer: 8 byte)
                optional program state information
                { integer, integer, ... }
                .
                .
end marker:     -1
```

The event type identifier is simply a sequential integer value, starting at 0 that corresponds to the list of events given in the user specification. *Graze* uses this integer value to tag the event record in the log file rather than using the actual string name for both space efficiency and to minimize conversion problems caused by executing on one platform and displaying on another. The resolution of the timestamp depends on the operating system and hardware. On the Solaris system, a system call, **gethrtime()**, returns the current wall clock time. This value is expressed as nanoseconds since some arbitrary time in the past; it is not related to user CPU time or system CPU time as returned by other system calls such as **rusage()**.

3. Visualization Tools

Graze provides two tools for generic data visualization: `gorge` and `nibble`, that can be used once the performance data has been collected. *Gorge* displays the collected data with wall clock time increasing along the x-axis as shown in Figure 2. `Nibble` allows the user to graph generalized functions of statistical information pertaining to specific events and intervals as shown in Figure 3. Both visualization tools perform the following processing on the raw data: (1) they combine all of the data from the log files into a single event stream in memory; (2) they sort the event

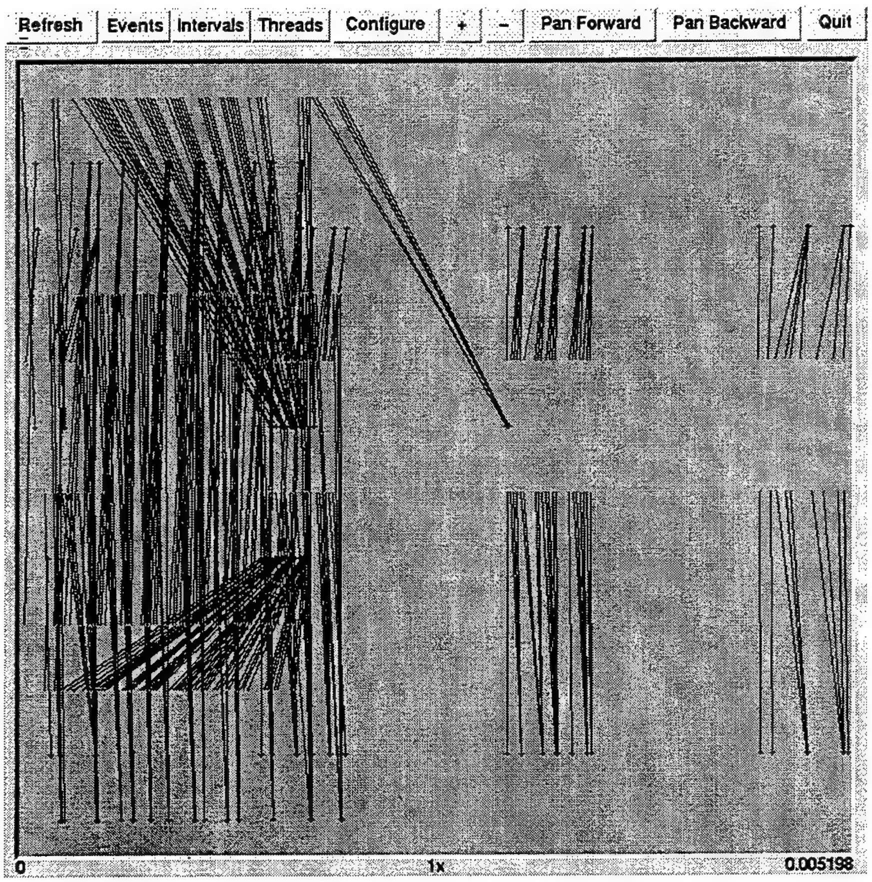

Figure 2. Gorge can be used to display the interaction between threads in a message passing application. For this example, event data from a total of 12 threads are shown, with event data from thread 0 plotted at the top of the graph and event data from thread 11 plotted at the bottom of the graph. The connecting lines between threads indicate the previously defined `Transit` interval. Time increases along the x-axis.

stream by time; and (3) they normalize the timestamp values so the first event in the event stream has the value of 0. Once the data have been preprocessed in this way, the visualization tools generate interval instances from the event stream by applying the matching conditions as defined in the user specifications.

As shown in Figure 2, the `gorge` tool displays data using a time-space graph, where the horizontal axis is the normalized time and the vertical axis represent data from each threads. The user can control the amount of information displayed by zooming in or out on the graph, and by selecting particular event, interval, and

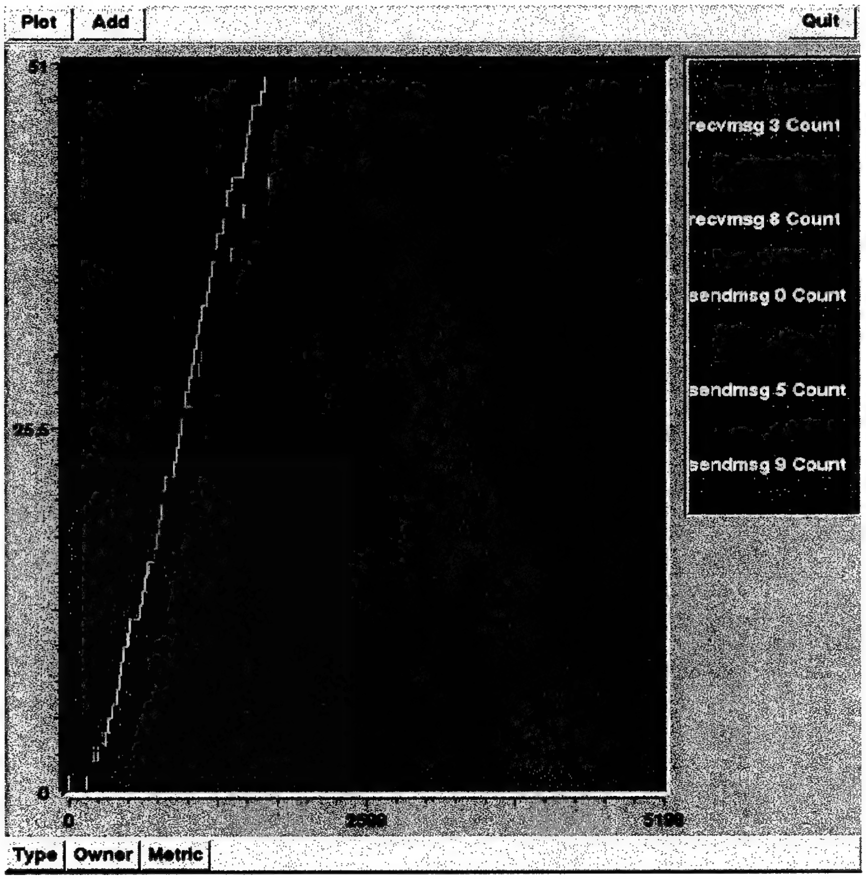

Figure 3. Nibble graphing the number of `Sendmsg` and `Recvmsg` events as a function of time in a message passing application. Time increases along the x-axis.

thread combinations [Ref. 1]. The `gorge` tool can provide a visual representation of the interaction between threads (Figure 2) by displaying intervals as a group of inter-connected lines.

As shown in Figure 3, the `nibble` tool is a generic statistical graphing tool for plotting data versus time. The types of data that can be graphed include quantities such as the total number of occurrences of an event or interval, a histogram of an event's state attributes, and the average elapsed time of an interval (Figure 3). The `nibble` tool provides mathematical operators that can be applied to the event or interval data to calculate new statistical data types on the fly for display. For example, graphs can be added, subtracted, multiplied, divided, and smoothed (Figure 4).

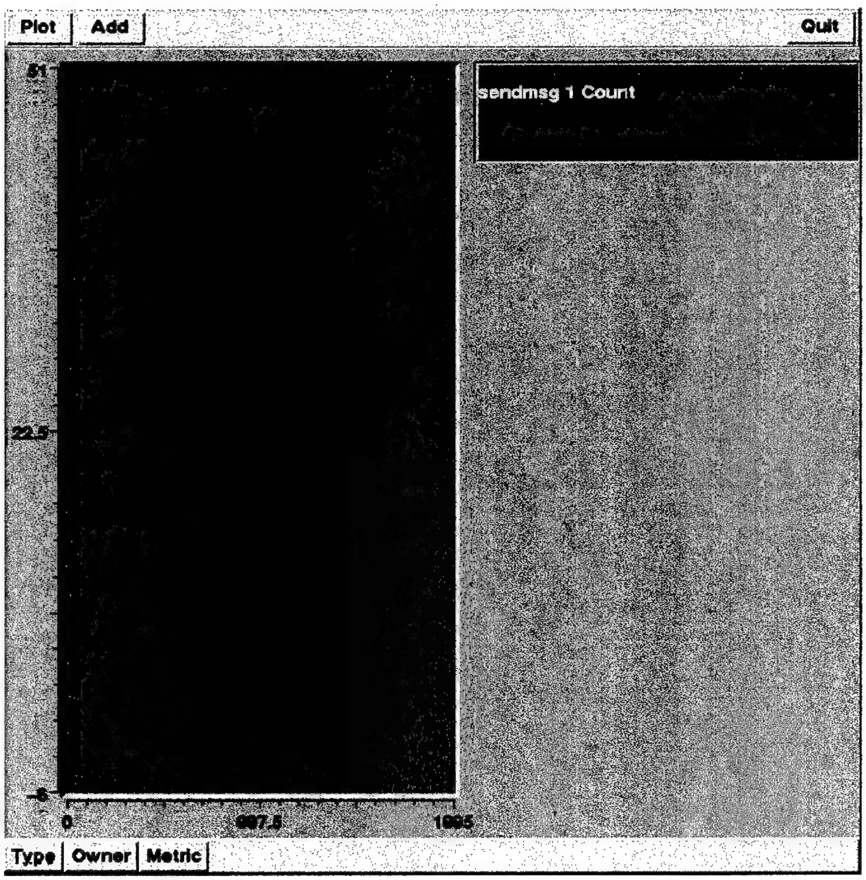

Figure 4. Nibble showing the number of `Sendmsg` and `Recvmsg` events in thread 0 and 1, and the differences (the lower line graph) between the number of messages received by thread 1 and number of messages sent from thread 0. Time increases along the x-axis.

4. Summary

In this chapter, we describe the three components of *Graze*: the specification language, the Data Collection Facility, and the visualization tools. The specification language lets the user defines `event` and `interval` types that are of interest for a given application. The specification defined by user is used to drive Data Collection Facility and visualization tools. In particular, the `event` specifications are automatically translated into data logging functions by the Data Collection Facility. The programmer then annotates the application with the `event` point by inserting data logging functions at the appropriate locations in the program. The data collected

from running the instrumented program, along with intervals constructed from the event data and the user-provided specification, can be plotted and analyzed by `nibble` and `gorge` tools.

IV. GLIMPSE: GENERALIZING GRAZE AND APPLYING IT TO MULTI-THREADED JAVA PROGRAMS

This chapter describes the design and implementation issues faced when generalizing *Graze's* Data Collection Facility. In particular, it addresses those issues faced when attempting to monitor multi-threaded Java programs (Appendix A).

The first section of this chapter provides an overview of the design issues encountered when generalizing *Graze* to apply to Java programs. These issues include mapping Java threads to event logs, accessing a high-resolution timer from within a Java program, and generalization of the *interval* definition. The next section describes event collection code generation, where *Graze's* events are translated into Java methods. The final section summarizes the design issues faced in implementing *Glimpse*.

1. Design Considerations

As described in the previous chapter, *Graze* provides a utility program that reads the user's specification and automatically generates C++ functions. The user then manually inserts events into the application. At execution time, event information is written to a log file. *Glimpse* provides similar capability for event information collection to Java programs. The following design goals and constraints were considered when implementing the *Glimpse* data collection facility:

i) We want to re-use *Graze* visualization tools; this implies maintaining the same data file structure and log filename convention in order to be compatible with the existing visualization programs.

ii) The overhead incurred due to the data collection code should be kept to a minimum. The modifications to the monitored programs should consist of a method call to initialize the data collection package, method calls to write out event data,

and a method call to free up resources used by the data collection package prior to program termination.

iii) To improve code modularity, the data logging methods, which depend on the user specification and are called (referenced) directly by the application program, should be placed in a separate module from the rest of the data collection package. This separation has the advantage that if the types of events to be monitored are changed, only the module containing data logging methods needs to be re-generated and re-compiled.

Based on these criteria, we implemented a new front-end program, **builder_java** and a new utility library, **glimpse.jar**. The **builder_java** program parses the specification and generates a Java class that contains data logging methods. The **glimpse.jar** library contains utility functions such as those needed for opening the log files, writing to the log files, and obtaining timestamp information. Several issues arose as we implemented the *Glimpse* data collection facility. The issues, and approaches we adopted, are described in the following sections.

a. Thread Naming

Graze stores event data from each thread in a thread-specific log file. To keep track of the log file that a data logging function is writing to, *Graze* uses the thread identifier provided by the system. The value of the thread identifier, which is of type integer, is obtained at runtime by calling a SUN Solaris thread library function. Based on this integer number, *Graze* determines the appropriate log file to which the event data is written. For every thread from which logging functions are called, there is a corresponding log file with the name of "log.{thread id}". This naming scheme is also shared by *Graze's* visualization tools, which expect data files to have names such as **log.2**, where **log.2** contains event data for thread 2, and where the log file for the first thread must be **log.0**.

This filename convention assumes that the system thread library uses a sequential numbering scheme, starting at 0, to keep track of threads that are created

by the program. It also assumes that threads used by the application exist for the duration of the program execution. By default, the Java thread library uses a string value such as "Thread-6" to represent a thread's unique identifier. The initial value of the integer in this string is system dependent. In some implementations of the JVM this number starts at 1 (Win95, Sun Java version 1.1.5), and in others it starts with 4 or 5 (Silicon Graphics IRIX 6.2, Java version 1.1.6). Furthermore, many threads in Java programs are short-lived. These are transient threads, created to perform a specific task in the application program. Once that task is completed, the thread exits the method it was created to execute and is garbage collected by the Java Virtual Machine. Threads created to perform asynchronous input and output operations are an example of short-lived threads.

Because threads in Java are inexpensive to create and destroy, an application program can potentially use a large number of transient threads. After a transient thread has exited, the Java thread library can re-use that same thread name again for a newly created thread. Directly translating *Graze* code to Java could therefore result in event data from two different threads being saved to the same log file. To resolve this thread names issue in a way that is consistent with *Graze's* current log filename convention so the existing visualization tools can read the data without modification, we designed and implemented a mechanism to associate each unique Java thread with its own log file.

One possible solution we considered was to require the monitored program to explicitly identify each thread that it creates with a unique integer, starting at 0. This approach will certainly prevent two threads from having the same name and thus saving event data to the same file. However it would mean calling the **setName(int id_value)** method (from the **Java Thread Library**) every time a thread is created in the application. This approach would likely add additional code to the monitored program as most programs do not explicit set the thread name but rather let it default. Additionally, this solution shifts the task of creating and managing the

unique thread name to the programmer. Therefore, this solution would make using the *Glimpse* Data Collection Facility unnecessarily tedious.

The approach we eventually adopted was to provide a mechanism for mapping the Java threads to unique log files that is transparent to the user (See Figure 5). The thread name to log file mapping is implemented using two utility classes: `ThreadData` and `ThreadPool`. The `ThreadPool` object maintains a pool of `ThreadData` objects. The `ThreadData` object contains various thread-specific information such as the actual name of the thread and the associated log filename. It also contains methods for initializing the log file, writing data, and closing the log file. With this solution, each time that an event logging method is invoked, that method sends a request to the `ThreadPool` object. The `ThreadPool` object returns a reference to the `ThreadData` object assigned to the thread that invoked the event logging method. If the `ThreadData` object for the calling thread is not found, then a new `ThreadData` object is created and added to the pool, and the reference to that newly created object is returned. The event logging method can then invoke the output methods of the `ThreadData` object to write out the corresponding timestamp and the user-defined event attributes to the appropriate log file. Any future write requests by the same thread will result in the `ThreadPool` returning a reference to the designated `ThreadData` object.

To ensure that every `ThreadData` object is assigned to a different log file, the `ThreadData` class maintains a global integer counter that is shared by all objects of this class. This counter is initially set to zero, and its value is incremented by one when a new `ThreadData` object is created. To ensure thread-safe behavior, the action of obtaining and incrementing this counter value by a thread is mutually exclusive from other threads that perform the same action. This mapping scheme isolates the thread name used by the Java program from the actual log filename, and preserves the original log filename convention.

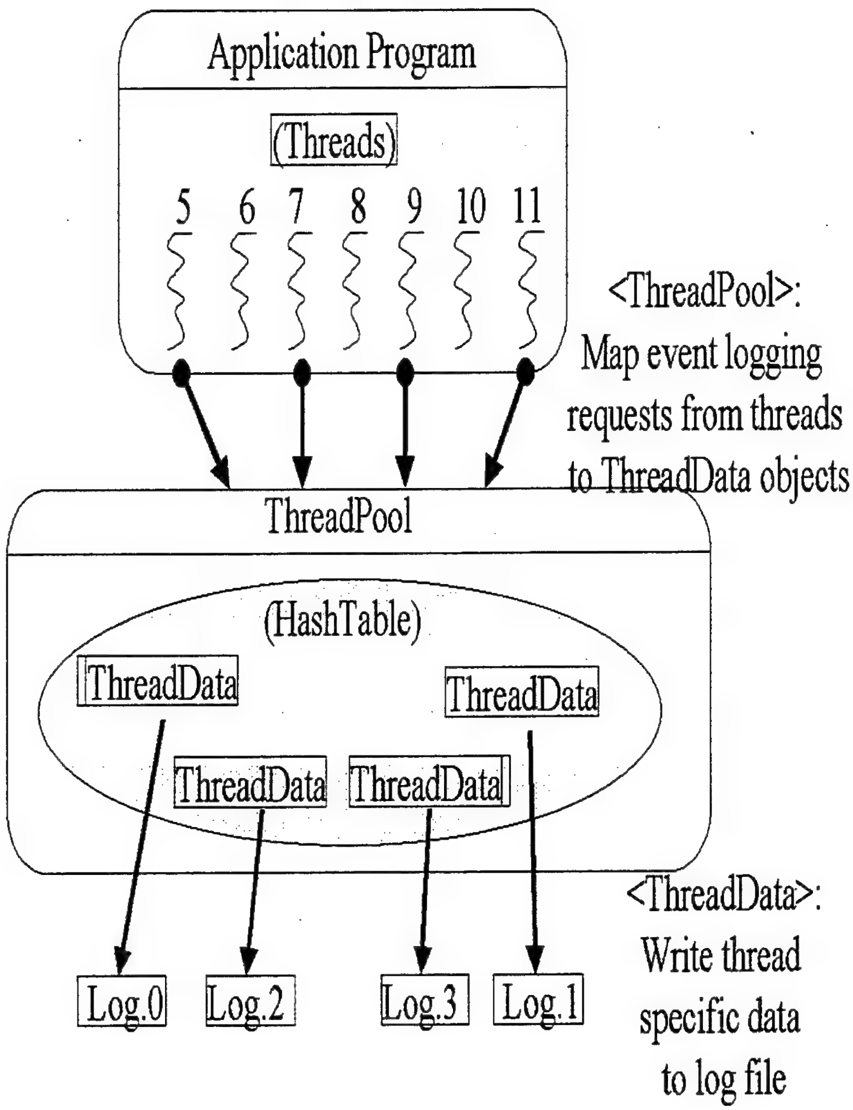

Figure 5. Glimpse's Java utility classes for mapping threads to log files.

To prevent saving event data from two different threads to the same file, as in the cases of short-lived threads with duplicate thread names, the only action required by the application is for it to notify the `ThreadPool` object when a transient thread is terminated. This notification is accomplished through the `closeThreadData()` method that is executed immediately before a thread terminates. This method removes the corresponding `ThreadData` object from the pool. Then, when another thread with the same name appears, the `ThreadPool` object will create a new `ThreadData` object and assign it a different log filename. [1]

b. *High Resolution Timer*

The Java core package provides a timing routine `currentTimeMillis()` in the `java.lang.System` class. This method returns the time, in milliseconds, between the current time and the standard base time known as "the epoch," 00:00:00 GMT on January 1, 1970. For performance monitoring with Graze, we need to record the real time at which events occur during the program execution. For some applications, millisecond resolution of the standard Java timing routine is too coarse. We investigated other alternatives for implementing a high resolution timer. One package we looked at is the PortableTimer from the **PTOOLS** working group [Ref. 16]. The PortableTimer specifies a set of timing functions that vendors should provide on their respective systems to facilitate performance measurement, and a sample implementation of the specification for several of UNIX platforms. The sample implementations use the standard UNIX functions to obtain wall clock, user and system time. Unfortunately, implementations of this specification are not yet available for our platforms.

Consequently we investigated building a custom timer. On the platform (Silicon Graphics R10000 architecture) where we did most of the development and

[1] We note that in the current implementation of Glimpse, that the user must explicitly call `closeThreadData()`. A commercial version of *Graze* could incorporate this call into the Garbage Collector. Another possible solution would be for the Java Thread API to provide a way to register a function that is invoked when a given thread exits. The `ThreadData` object would then automatically register a `closeThreadData()` method, and thus eliminates the need for the user to explicitly call this method.

code testing, a high resolution timing function is supported via a free-running 64-bits hardware counter. To use this hardware counter for performance timing, we wrote a custom timer that maps the location of the hardware counter to an address in the user process space. The value of the hardware counter can then be obtained by simply reading the value stored at that address. Depending on the particular version of the R10000 architecture, the hardware counter has a resolution (the elapsed time between ticks) between 21 and 800 nanoseconds. This resolution value can be dynamicallly determined by querying the hardware at run time.

To access this custom timer (which is implemented in the C language) from a Java program, we use the Java Native Interface (**JNI**) [Ref. 17]. JNI provides a standard mechanism for a Java program to access functions in machine binary code. The basic steps of using JNI are: (1) write a Java class that declares the C functions, with the appropriate return types and calling arguments and a keyword 'native' before the method name; (2) use the Java utility tool `javah`, to translate these Java methods to the equivalent C-language function prototypes and store them in a header file; (3) provide the implementation (in C or C++) of these functions as declared in the header file; and (4) create a dynamic shared library of these functions, i.e., the **.dll** library in the Windows NT environment or **.so** library in the UNIX environment. For example, the high-resolution timer used in our tests was implemented as follows:

```
final public class SystemTimer
{
..
/* Name of native function (implemented in the C-lang) that
   returns time in nanosecond.  The implementation of the
   function gethrtime_ns() must be provided in a dynamic
   shared library.  In this example this library is called
   libsgitimer.so . */
      public static native long gethrtime_ns();

/* Static Initializer: Load the dynamic shared library.   */
      static {
          System.loadLibrary("sgitimer");
```

```
    }
/* SystemTimer method that returns real time in nanosecond.
   It calls the native function gethrtime_ns().  */
      public static long gettime_ns()
      {
          return gethrtime_ns();
      }
..
}
```

The keyword `native` before the `gethrtime_ns()` method declaration tells the JVM that the actual implementation of this method in not in Java bytecodes, but machine binary codes. When the `SystemTimer` object is initialized, the JVM executes `System.loadLibrary("libname")` to load the dynamic shared library containing the high-resolution timer functions. The method `SystemTimer.gettime_ns()` simply calls the equivalent native function to obtain the real time in nanoseconds.

c. *Generalization of the Interval Definition*

We recall from the previous discussion of *Graze* in the last chapter that an *interval* is determined by two bounding events, and by the matching criteria, if specified, between the start and end events. The two criteria are to require that both events occured on the same thread, or to match all event attributes (except for the thread identifier and timestamps) of the two bounding events. For example:

```
event Sendmsg(src, seq) = plus;
event Recvmsg(src, seq) = box;
interval Transit1 [ s:Sendmsg -> r:Recvmsg ] = line;
interval Transit2 [ s:Sendmsg -> r:Recvmsg ] Match = line;
```

Interval `Transit1` requires that both `Sendmsg` and `Recvmsg` events occur in the same thread; interval `Transit2` requires that (`src` and `seq`) values are the same. We would like to generalize the *interval* definition so that it allows specification for partial matching of attribute values, and Boolean relationships between the event's attributes. This new syntax would support the following types of interval definitions:

```
interval Transit3 [ s:Sendmsg -> r:Recvmsg ]
        { s.src==r.src } = line;

interval Transit4 [ s:Sendmsg -> r:Recvmsg ]  = line;
        { s.src==r.src && s.seq >= r.seq } = line;

interval Transit5 [ s:Sendmsg -> r:Recvmsg ]  = line;
        { s.src!=r.src && s.seq < r.seq } = line;
```

Interval `Transit3` requires only the `src` variable of `Sendmsg` and `Recvmsg` to match. Interval `Transit4` requires the `src` value to match and that `seq` value of `Sendmsg` event is greater than or equal to the `seq` value of `Recvmsg` event. Under this expanded syntax (see Appendix B), the operators for comparing the attribute values are:

$$>, >=, ==, <=, <, ! = \tag{IV.1}$$

To support this more general form of interval specification, additional production rules are added to the lexical parser of the *glimpse*, and the internal data structure used by the semantic analyzer is expanded to include the additional clauses for evaluating the attribute data between the two events.

2. Event Collection Code Generation

We created a front-end program, **builder_java**, to automatically create application specific Java class that contains data logging methods. For each event type defined in the user specification, **builder_java** creates a corresponding static method. As an example we consider the specification for message passing events:

```
event Sendmsg(src, seq)
```

and

```
event Recvmsg(src, seq)
```

The Java class, and its methods, created from the above specification are:

```
final public class GzEvent
{
  static ThreadPool pool;
  final static boolean DISABLE_GLIMPSE=false;
  final static boolean NATIVE_TIMER=false;
  static SystemTimer st;

  static public void init()
  {
    if (DISABLE_GLIMPSE) return;
    pool=new ThreadPool();
    if (NATIVE_TIMER==true) st=new SystemTimer();
  }
  static public void init(String data_dir)
  {
    if (DISABLE_GLIMPSE) return;
    pool=new ThreadPool(data_dir);
    if (NATIVE_TIMER==true) st=new SystemTimer();
  }
  static public void close()
  {
    if (DISABLE_GLIMPSE) return;
    pool.closeAll();
  }
  static public long gettime()
  {
    long ts;
    if (NATIVE_TIMER)
        ts=st.gethrtime_ms();
    else
        ts=System.currentTimeMillis()*1000;
    return ts;
  }
  static public void Recvmsg(int src, int seq)
  {
    if (DISABLE_GLIMPSE) return;
    long ts=GzEvent.gettime();
    ThreadData handle= (ThreadData) pool.getThreadSpecific();
    handle.putout(0);
    handle.puttime(ts);
    handle.putout(src);
    handle.putout(seq);
```

```
    return;
  }
  static public void Sendmsg(int src, int seq)
  {
    if (DISABLE_GLIMPSE) return;
    long ts=GzEvent.gettime();
    ThreadData handle= (ThreadData) pool.getThreadSpecific();
    handle.putout(1);
    handle.puttime(ts);
    handle.putout(src);
    handle.putout(seq);
    return;
  }
  static public void closeThreadData()
  {
    if (DISABLE_GLIMPSE) return;
    pool.closeThreadData(Thread.currentThread());
  }
}
```

The name of the generated event logging class defaults to GzEvent, although the user can specify a different name by passing an optional argument to the **builder_Java** program. Calls to the `SystemTimer` methods to get timestamp values, and to write out the values of variables `src` and `seq` are automatically inserted into the body of the event logging methods, `GzEvent.Recvmsg(src,seq)` and `GzEvent.Sendmsg(src,seq)`. A flag, `NATIVE_TIMER`, can be set to allow the logging methods to use either the standard Java timing method, or the native timing function, if one is provided via the JNI mechanism as discussed in the previous section. To instrument the application, the programmer edits the source code and annotates it using `GzEvent.init()`, `GzEvent.Recvmsg()`, and `GzEvent.Sendmsg()`. [2] Two versions of the `GzEvent.init()` methods are provided–one takes an optional argument to specify the directory to which the log files are to be written. If not specified, the current working directory of the monitored program is used. As stated earlier, the

---

[2]In the current implementation the programmer must also manually insert a call to the `GzEvent.close()` method. See appendix C on using Java Finalize facility.

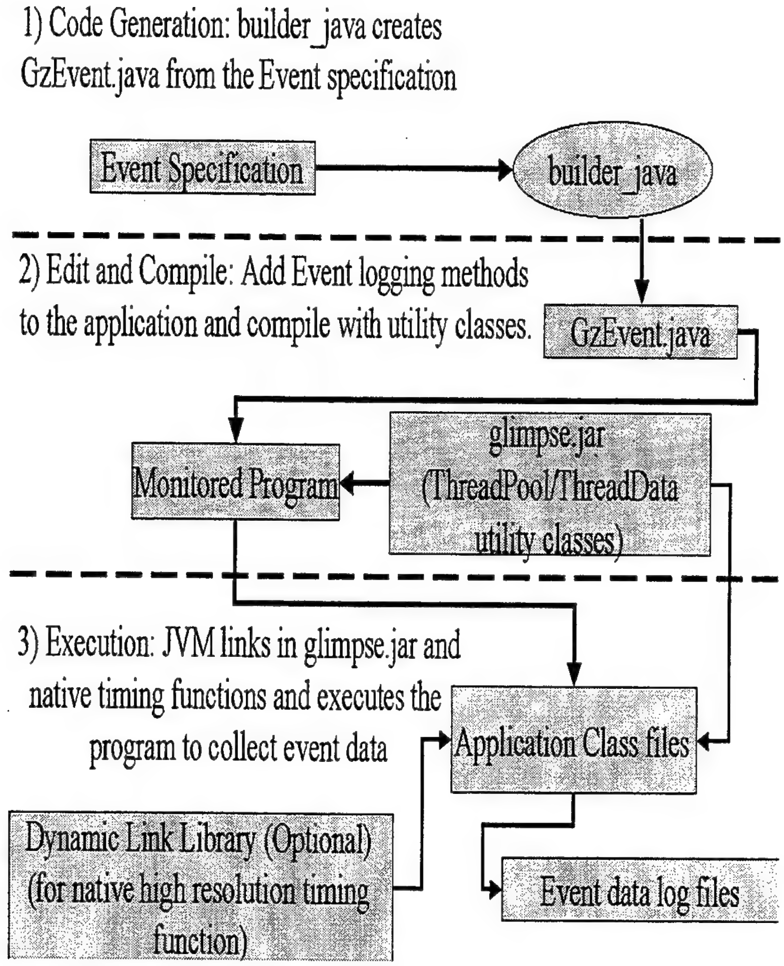

Figure 6. Steps for collecting event data: 1) code generation, 2) compiling the monitored program, 3) loading dynamic library during execution. Arrow indicates dependency at the various stages.

`GzEvent.closeThreadData()` method notifies the ThreadPool object that the calling thread is about to exit and that the log file associated with the exiting thead should be closed. This call is only needed to handle tranisent threads; it prevents data from two different threads being written to the same log file. Finally, `GzEvent.close()` terminates event logging and closes all log files. A flag, `DISABLE_GLIMPSE`, allows the user to turn off the data logging if the application no longer requires performance monitoring.

To compile the monitored program, both the `GzEvent.java` file and `glimpse.jar` utility library need to be supplied to the Java compiler (See Figure 6). The `glimpse.jar` library and dynamic shared library for the native high-resolution timer are then loaded in by the JVM during the execution of the monitored program.

3. Summary

In this chapter, we described the porting issues encountered while adapting *Graze* to monitor multi-threaded Java programs, and the approaches we used to resolve these issues. In particular, the issues with mapping application threads to event log filename, obtaining high-resolution timestamp information at run-time, and generalizing the *interval* definition. In the last section, we descibed how *Glimpse* translates user-defined event specification into the data logging code that is used to instrument the application of interest.

THIS PAGE INTENTIONALLY LEFT BLANK

V. EXPERIENCES USING GLIMPSE WITH JAVA

In this chapter we describe the result from testing *Glimpse* with a multi-threaded Java program. By visualizing the event data collected from execution of the test program and intervals derived from the event data, and by relating these events and intervals to the actions performed by a test program, we demonstrate that *Glimpse* is working properly.

The first section of this chapter describes the structure of a Java example program and the expected interactions between the various threads in this application. The next section describes the event and interval specifications that were used to collect the runtime data, and to visualize that data. Section three describes the data gathered from running the test program with two different inputs. By relating the events and intervals to the expected behaviors of the application, we demonstrated the correctness of the test program and *Glimpse's* Data Collection Facility. In the final section, we summarize the results from testing *Glimpse* with a multi-threaded Java program.

1. Description of the StopLight Program

To evaluate the *Glimpse* software, we applied it to a Java program called *"StopLight"* that simulates traffic flow at an intersection. This program was originally written to show how to use Java threads in a concurrent program. It uses car, intersection, and timer objects to model the flows through the intersection. Each car object is assigned a direction of movement (North-South or East-West) and a departure time. The departure time determines when a car will arrive at the intersection. The timer object controls the traffic lights of the intersection object. Each timer object is initialized with a timeslice value. When that timeslice value expires, the timer object toggles the traffic lights so that the flow direction at the intersection alternates between North-South and East-West. Two pair of timers are used to

control the traffic light at the intersection; their function will be described in next paragraph. Cars traveling in different directions can arrive randomly at the intersection, and more than one car can queue up at the intersection while waiting for the light to change. In this program, a car is not allowed to change direction once it starts running; furthermore there is only one intersection through which all cars must travel.

An intersection should maximize the number of cars passing through, and minimize the time that a car is waiting at the intersection. For example, the green light should stay on longer for the direction with a heavy traffic flow, and should turn red quickly if no more cars are passing through. To implement this behavior in the StopLight program, a timer called LargeTimer (that is a timer with a longer expiration value) and another timer called SmallTimer are used concurrently to control the traffic light. Both timers are started when the intersection object is first initialized. Each time a car passes through the intersection without stopping, the currently active SmallerTimer is stopped and a new one is started. Stopping and restarting this SmallTimer has the effect of prolonging the green light for that given direction. Thus, if a group of cars, traveling along the same direction, arrives at the intersection at close intervals, then the traffic light for that direction will tend to stay green. However, if there is a long gap between cars travelling along the same direction arriving at the intersection, then the SmallTimer will most likely expire before the next car can reach the intersection. When the SmallTimer expires it forces the traffic lights to change, thus allowing the cars traveling along the other direction to proceed. This control logic ensures that cars waiting at the intersection will not wait longer than necessary (not greater than the expiration value of the SmallTimer) if the cross traffic is very sparse.

The LargeTimer ensures that both directions are given their fair share of green light. For example, consider the case where there is a caravan of cars traveling along the North-South direction, while only a few cars are traveling along the East-West

direction. If the gaps between the vehicles in the caravan are very short and there were no LargeTimer object, it is likely then that North-South bound cars could keep extending the traffic light in their favor, and the East-West bound cars would not be able to proceed until all the North-South bound cars were through. The LargeTimer prevents this unfair situation from occurring. Regardless of the number of times that a SmallTimer is restarted, the LargeTimer will eventually expire and force the traffic light to switch. This design guarantees that the maximum waiting time of any cars will not exceed some threshold value, regardless of the uneven traffic distribution between the two flow directions.

In the implementation of the StopLight program, each car and timer object executes as a separate Java thread. Timer threads are assigned higher priority than car threads. The higher priority associated with the timer thread allows it to interrupt other threads in order to change the traffic light at the intersection. The critical sections of this program are associated with changing the traffic light variable of the intersection object. To ensure the synchronized access to the traffic light variable, the intersection object acts as the monitor to the critical sections. (See Appendix A for a discussion on the Java monitor object and synchronization.)

2. Defining Events and Intervals For the StopLight Program

To evaluate the *Glimpse* package with the Stoplight program, we define the following events that are relevant to the car, timer and the intersection objects. For the car objects, we defined three types of events: `StartMotor`, `ArriveIntersection` and `LeaveIntersection`. These events correspond to 1) when the car is started, 2) when it reaches the intersection, and 3) when the car has crossed the intersection. In term of actual location in the `StopLight` program code, these events correspond to 1) when the car object enters its `run()` method, 2) when the car object invokes the `enter()` method of the intersection object, and 3) when the car object returns from the `enter()` method of the intersection object.

For the intersection object we defined the `Stop`, `Go`, and `NoWait` events. The `Stop` event occurs when a car reaching the intersection has to wait for the light to change. The `Go` event occurs when a car waiting at the intersection can proceed. The `NoWait` event occurs when the traffic light direction and car direction are the same, in that case the car just proceeds through the intersection without stopping. The following code shows the locations of event collection points in the `Intersection` object:

```
public class Intersection {
    public int traffic_light ;
    public LargeTimer MajorTimer;
    public SmallTimer MinorTimer;
    . . .
    public synchronized void enter(Car cx, int car_direction)
    {
        /* car going at NORTH or SOUTH direction and so is the current
           traffic light for the intersection. */
        if ( traffic_light == NSDirection &&
             (car_direction == NORTH || car_direction == SOUTH) )
        {
             GzEvent.NoWait(traffic_light);
             MinorTimer.stop();
             MinorTimer = new SmallTimer(200,this);
        }
        /* car going at EAST or WEST direction and so is the current
           traffic light for the intersection. */
        else if ( traffic_light == EWDirection &&
                  (car_direction == EAST || car_direction == WEST) )
        {
             GzEvent.NoWait(traffic_light);
             MinorTimer.stop();
             MinorTimer = new SmallTimer(200,this);
        }
        /* car direction and traffic light direction does not match,
           car must wait until the light changed. */
        else {
            try {
                GzEvent.Stop(traffic_light);
                wait();
```

```
                GzEvent.Go(traffic_light);
            } catch (Exception e) {}
        }
    }
}
```

In the `enter()` method of the Intersection object, the first two '`if`' statement blocks handle the cases where the car direction matches the traffic light direction. In these two `if` statement blocks, the old `SmallTimer` is killed and a new `SmallTimer` is started as per discussion in section one, and execution continues without blocking. In the third `if` statement block, the execution of the `enter()` method is blocked on the call to the `wait()` method. This car will remain blocked until another thread, i.e., the timer thread, acts on the `traffic_light` variable and invokes the `notifyall()` method. (See Appendix A on the `wait()` and `notifyall()` mechanism for synchronizing access to shared variables from multiple threads.)

For the timer objects, we define the `LightChanged` event. This event occurs when the timer acts on the traffic light variable of the Intersection object. We also defined the `SmallTimer` and `LargeTimer` events to indicate when a timer is instantiated. The following code shows the locations where we place the `LightChanged` event in the `LargeTimer` object:

```
public class LargeTimer extends Thread {
    private int timeslice;
    private Intersection in;
    . . .
    public void run()
    {
       try
       {
           /* Sleeping until timeslice expired */
           sleep(timeslice);
           /* Enter critical section and switch the traffic light
              variable of the Intersection object.  Notify any other
              threads waiting on the Intersection object that
              traffic light has changed. */
           synchronized (in)
```

```
            {
                in.MinorTimer.stop();
                if (in.traffic_light==NSDirection) {
                    in.traffic_light = EWDirection;
                    GzEvent.LightChanged(EWDirection);
                    in.notifyAll();
                }
                else
                {
                    in.traffic_light = NSDirection ;
                    GzEvent.LightChanged(NSDirection);
                    in.notifyAll();
                }
                /*  Reset all timers prior to exiting. */
                in.MajorTimer = new LargeTimer(800,in);
                in.MinorTimer = new SmallTimer(200,in);
                GzEvent.closeThreadData();
            }
        } catch (InterruptedException e) { }
    }
}
```

Upon the expiration of the timeslice, the timer (i) toggles the traffic light variable of the Intersection object, (ii) notifies any other threads waiting for the traffic light to change, and (iii) resets the Large and Small timers controlling the Intersection object before exiting.

The event specification for the `StopLight` program is shown below. Additional attribute information is recorded for some of the events. For examples, for events associated with the car objects, the direction of car movement is recorded; for events associated with the intersection and timer objects, the direction of the traffic light is saved to the log files.

```
graze msgs
. . .
event Stop(traffic_light_direction) = symbol(plus);
event Go(traffic_light_direction) = symbol(diamond);
event NoWait(traffic_light_direction) = symbol(x);
event SmallTimer(traffic_light_direction) = symbol(x);
```

```
event LargeTimer(traffic_light_direction) = symbol(plus);
event LightChanged(traffic_light_direction) = symbol(diamond) ;
event ArriveIntersection(car_direction) = symbol(x) ;
event LeaveIntersection(car_direction) = symbol(plus) ;
event StartMotor = symbol(box);
. . .
end msgs.
```

From the event definitions, we specify the following intervals to be constructed and visualized:

```
graze msgs
. . .
interval StartMotorArrInt [StartMotor, ArriveIntersection]=rectangle;
interval ArriveLeave [ArriveIntersection,LeaveIntersection]=rectangle;
interval ArriveAndStop [ArriveIntersection, Stop]=rectangle;
interval StopAndGo [Stop, Go]=rectangle;
interval GoAndLeaveInt [ Go, LeaveIntersection ]  = rectangle;
interval STimerLtChanged [ SmallTimer, LightChanged ] =  rectangle;
interval LTimerLtChanged [ LargeTimer, LightChanged ] =  rectangle;
interval LtChangedAndGo [ LightChanged <- Go] Match = line;
interval NoWaitSmallTimer [NoWait,SmallTimer] Match = line;
. . .
end msgs.
```

`StartMotorArrInt` shows the interval between when a car starts and when it reaches the intersection. The `ArriveLeave` interval depicts cars that arrive at the intersection and go through it without stopping, whereas `ArriveAndStop` interval shows cars that arrive at the intersection and wait for the light to changed. The interval `StopAndGo` shows how long a car spends waiting for the light to change. The `STimerLtChanged` and `LTimerLtChanged` depict the intervals between when the timer is started and when it actually acts to change the traffic light. The `LtChangedAndGo` and `NoWaitSmallTimer` intervals associate events occurring on different threads.

3. Visualizing the Result

We used two different traffic patterns to test the working of Glimpse with the StopLight program. The traffic patterns are configured by setting the departure

time and the direction of the cars. In test case one, cars are initialized with random departure time and different directions. In test case two, a stream of cars traveling along the same direction is started at fairly close intervals. This steady stream of traffic flow is interspersed with a few cars traveling along the other direction. From the previous discussion on the StopLight program, we would expect that in test case one, where cars arrive randomly at the intersection while traveling at different direction, it is more likely that we should see the SmallTimer acts to switch the traffic light. In test case two, a steady flow of cars traveling along the same direction arriving at the intersection will tend to prevent the SmallTimer from changing the traffic light, but will favor the LargeTimer.

We ran the `StopLight` program with the two test cases multiple times on a SGI workstation (with the Java SDK 1.1.6 environment). The results vary slightly from run to run, but the general pattern is consistent between different runs. The following plots show the representative results.

a. Experiment One

In Figure 7, the interval `StartMotorArrInt` are plotted using the *gorge* visualization tool. The symbol `x` at the right edge of the horizontal bar indicates when a car arrives at the intersection. Because of the different departure times assigned to the cars, car objects instantiated later in the main program can reach the intersection earlier. For example, car 4 reaches the intersection before car 3.

In Figure 8, we added the `StopAndGo` interval to the plot. The `StopAndGo` interval depicts a car that must stop at the intersection for the light to change. From the plot, we see that car 1, 3, 5, 9, 10, 12 stopped at intersection before they were allowed to continue through. Next, we add the events associated with timer objects to the plot. The events `LargeTimer` and `SmallTimer` are depicted by the symbol + and the symbol `x` respectively. The first occurrence of a pair of `LargeTimer` and `SmallTimer` are associated with the instantiation of the `Intersection` object. These two events are shown at top of the plot (Figure 9) before the horizontal bars

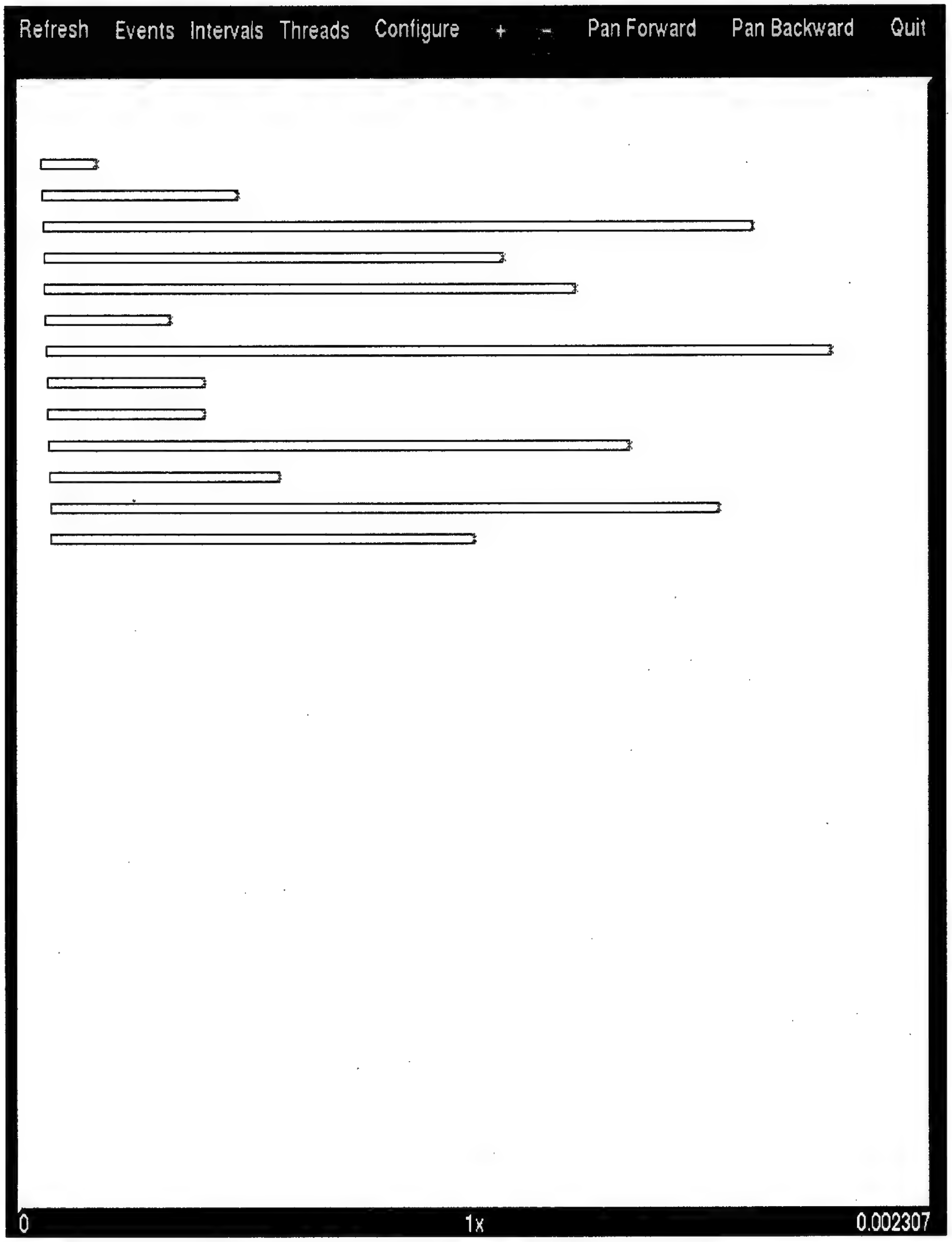

Figure 7. Plot of `StartMotorArrInt` intervals for experiment one. The horizontal bar indicates the interval between when a car is started and when it reaches the intersection. The time scale increases to the right. There are 13 cars in this test case; they are displayed from top to bottom.

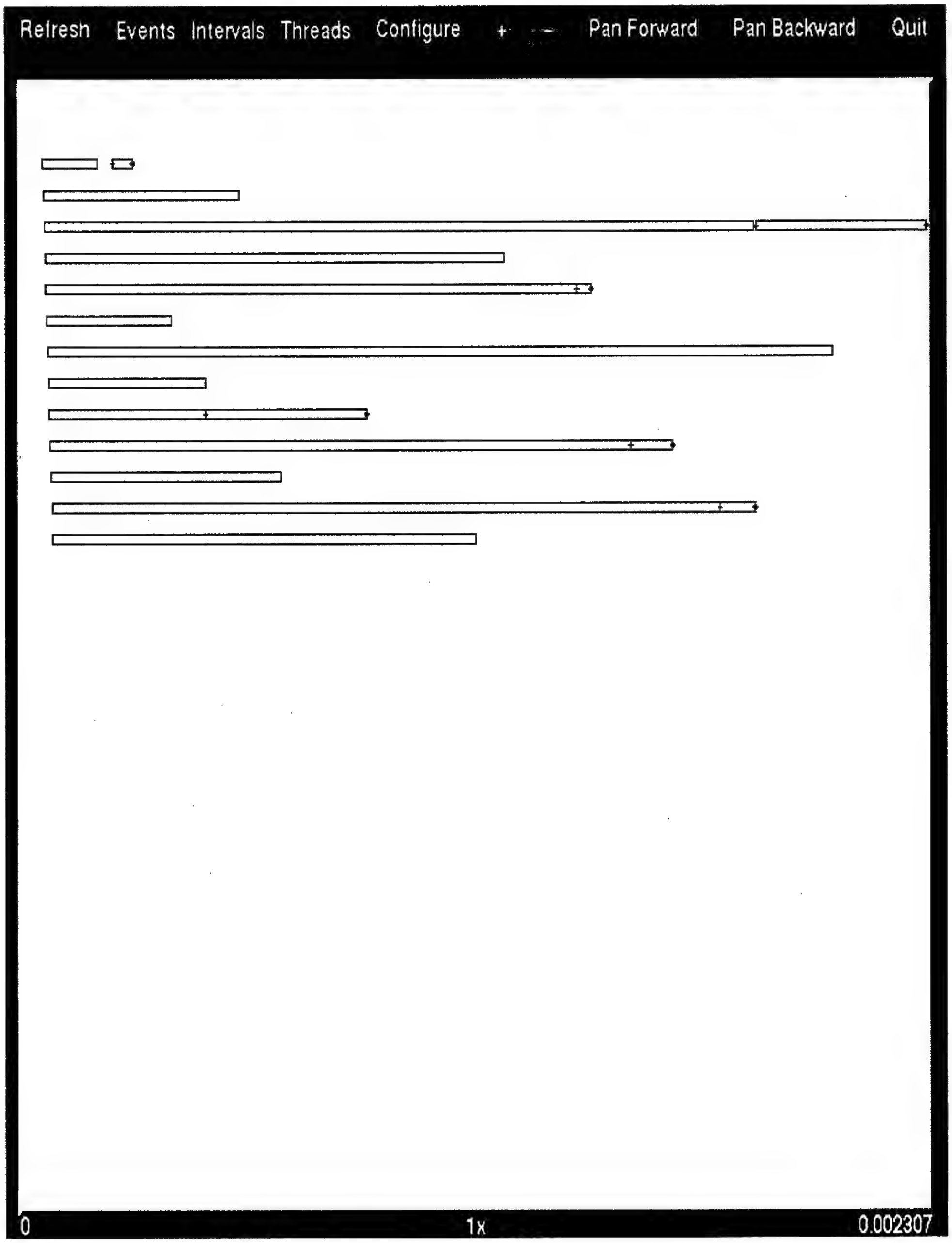

Figure 8. Plots of the **`StartMotorArrInt`** and **`StopAndGo`** intervals for experiment one. The StopAndGo intervals are bound by **`+`** and **`diamond`** symbols. Car 1, 3, 5, 9, 10, 12 have **`StopAndGo`** interval.

associated with car objects. Subsequently, we saw more instances of the `SmallTimer` event further down on the plot. The occurrences of these `SmallTimer` events are made clear in the next figure.

In Figure 10, we added the interval `NoWaitSmallTimer` to the plot. The `NoWaitSmallTimer` interval is represented by the vertical line that connects the right edge of the `StartMotorArrInt` interval in a car thread to the symbol `x` denoting the instantiation of a new `SmallTimer` object in a timer thread. There are 7 `NoWaitSmallTimer` intervals originating from car 2, 4, 6, 7, 8, 11, and 13; they correspond to cars that did not have to wait at the intersection for the light to change. Recall from the discussion of the *StopLight* program, cars that go through intersection without stopping are extending the duration of the traffic light in their direction by resetting the controlling timers. The interval `NoWaitSmallTimer` corresponds to this action, and the *gorge* provides a visual representation of this program behavior.

In Figure 11, We add `StimerLtChanged` and `LtChangedAndGo` intervals to the plot. The `StimerLtChanged` represents the interval between when a small timer is created and when its timeslice value has expired, at which time the small timer changes the traffic light direction. The `StimerLtChanged` interval is depicted as horizontal bars bounded by the `x` and `diamond` symbols. These `StimerLtChanged` intervals are shown below the car threads in Figure 11, except for the first instance of `StimerLtChanged` interval which is created by the `Intersection` object. From the plot we see that not every small timer thread contains a `StimerLtChanged` interval. This is because some small timers are preempted by cars going through the intersection, as discussed in the previous paragraph. The `LtChangedAndGo` represents the interval between when a car object stops at an intersection and when it starts moving again as the result of the expiring timer changing the traffic light direction. The `LtChangedAndGo` is depicted as vertical line connecting the right edge of the horizontal bars of the `LtChangedAndGo` intervals to the right edge of the `StopAndGo` intervals in the car threads. As can be seen from Figure 11, `LtChangedAndGo` intervals

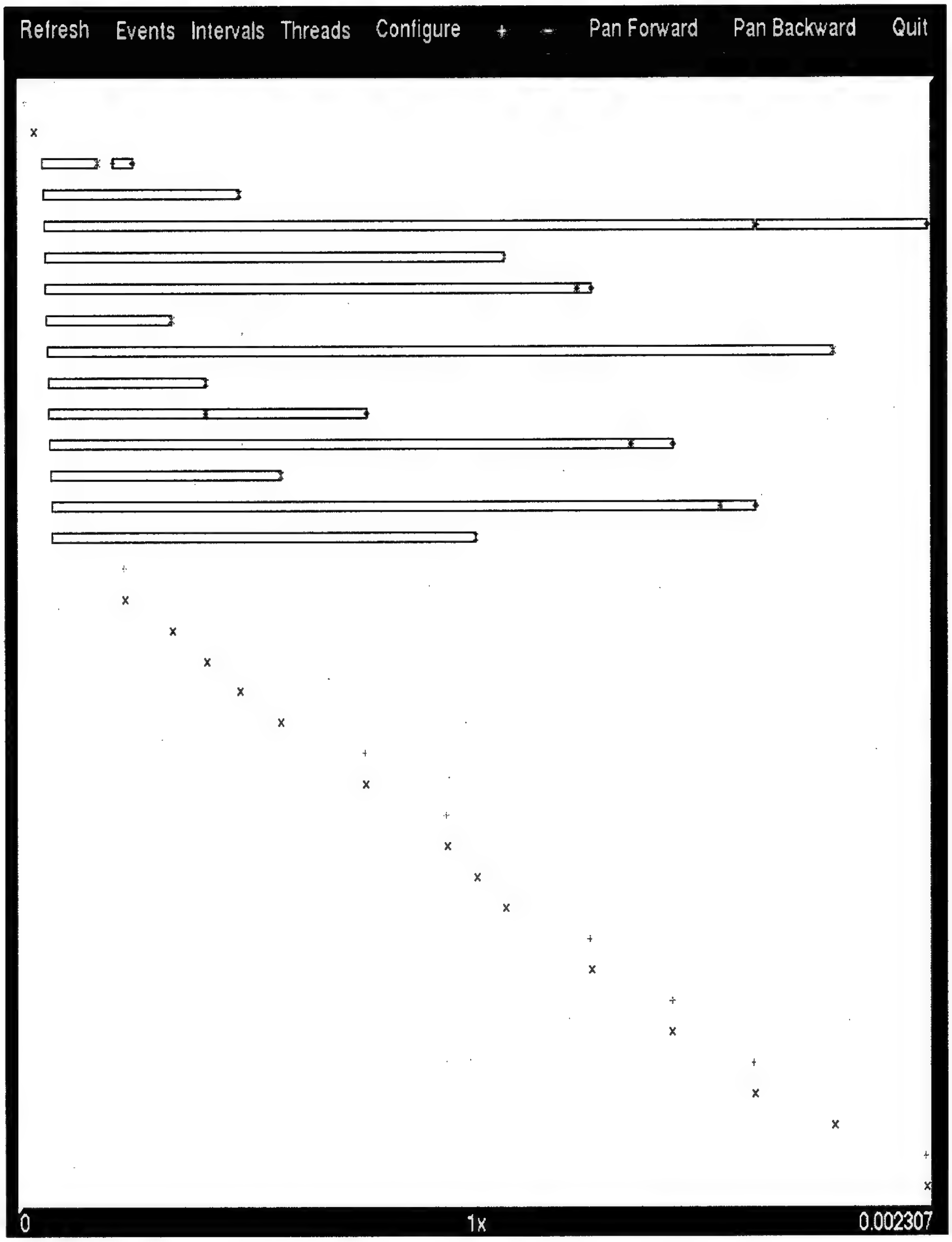

Figure 9. Plot of `LargeTimer` and `SmallTimer` events with `StartMotorArrInt` and `StopAndGo` intervals in experiment one. The symbol + and symbol x depict when the `LargeTimer` and `SmallTimer` objects are instantiated in the `StopLight` program. The first pair of timers is created (shown at the top of the plot) when the `Intersection` object is initialized, that is before any car objects are created.

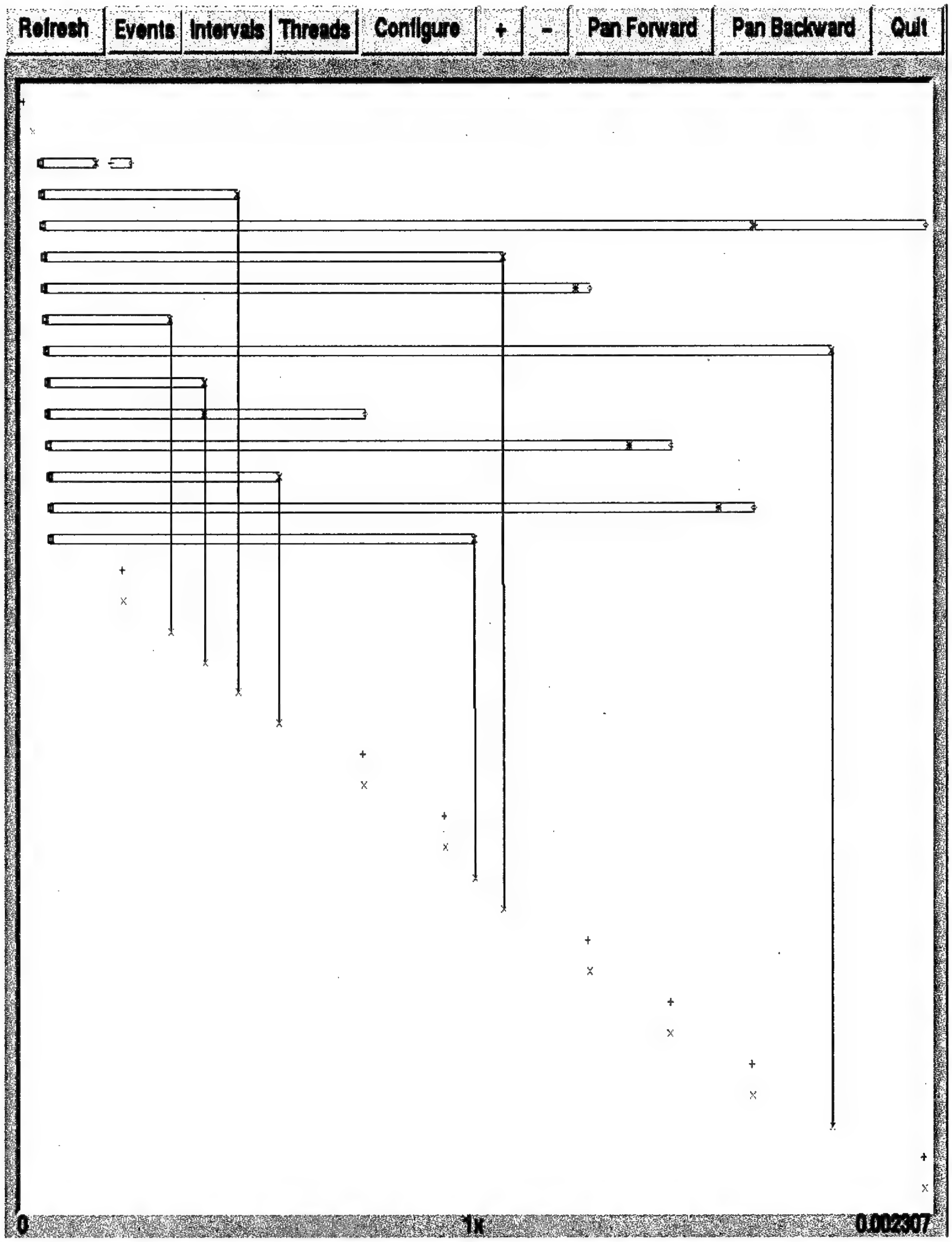

Figure 10. `NoWaitSmallTimer` interval is represented by the vertical line connection between the symbol `x` in the car thread and symbol `x` in the `SmallTimer`. There are 7 `NoWaitSmallTimer` intervals, originating from cars 2, 4, 6, 7, 8, 11, and 13. These intervals represent the new `SmallTimer` objects created by cars that did not have to stop at the intersection.

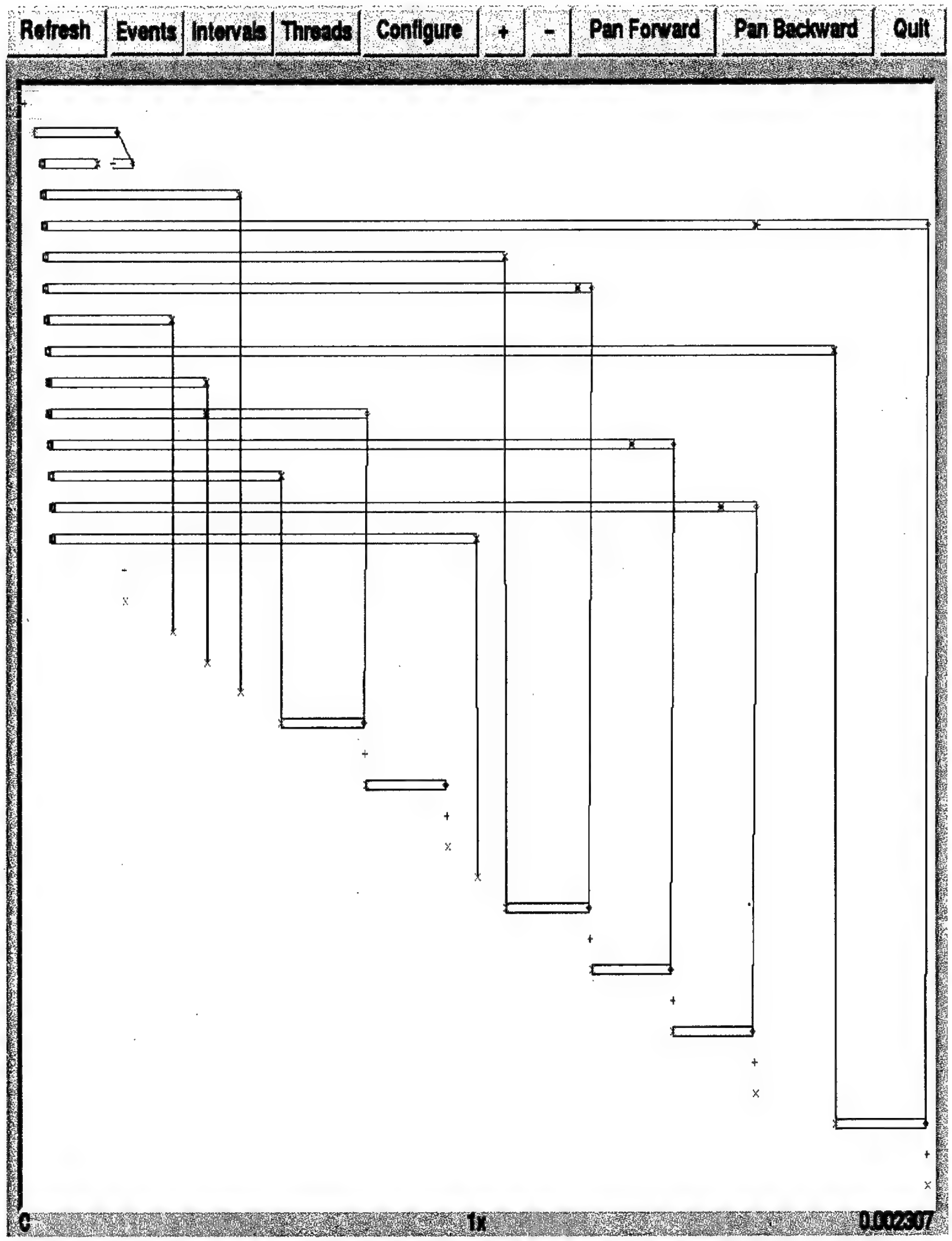

Figure 11. Interval `StimerLtChanged` is shown as the horizontal bar (between symbol `x` and symbol `diamond`) in timer threads. These intervals indicate the small timers that change the traffic light as the result of its timeslice value expiring. The interval `LtChangedAndGo` is shown as a vertical line from the right edge of the `StimerLtChanged` interval of the timer thread to the right edge of the `StopAndGo` interval of the car threads. These intervals represent the notification received by the car threads when the traffic light is changed by the timer threads.

connects car 1, 3, 5, 9, 10, and 12 to `StimerLtChanged` interval. These are cars that were waiting at the intersection for the light to change. The interactions between the timer threads and car threads are consistent with the traffic distribution used in this test case and the design of the *StopLight* program. The visualization of intervals and events allow us to correlate the execution of the application program with the semantics of the program code.

b. *Experiment Two*

In this experiment, we execute the annotated test program using a different traffic pattern. This traffic pattern consists of cars arriving at the intersection at a constant interval, as shown by the `StartMotorArrInt` intervals in Figure 12. In Figure 13, we added the `StopAndGo` interval to the plot. We see that cars 1, 5, 12, and 13 must stop at the intersection before proceeding.

In the next Figure (Figure 13), the events associated with the creation of `SmallTimer` and `LargeTimer` are shown. These events are associated with the car that did not have to stop at intersection. The interval `NoWaitSmallTimer` in Figure 15 clearly shows the relationship of car threads creating the new timer threads.

The difference between experiment one and experiment two is that in experiment two only one of the `SmallTimer` threads ever reaches expiration. For the experiment two, there is a steady stream of cars going through the intersection, so the exiting `SmallTimer`, except for the last one, is always being re-started. Recall that from the discussion of the *StopLight* program, if cars repeatedly passing through the intersection constantly extend the traffic light, at some point the `LargeTimer` will act to change the traffic light. This is indeed what is shown by the `LtimerLtChanged` interval in Figure 16. The LargeTimer expired and allowed car 5, which has been waiting at the intersection, to proceed. When the `LargeTimer` expired, it created a pair of `SmallTimer` and `LargeTimer`. The event `SmallTimer` and `LargeTimer`, and the interval `StimerLtChanged` are shown at the bottom right of Figure 16. The expiration of the last `SmallTimer` allows cars 12 and 13 to proceed. This is shown by

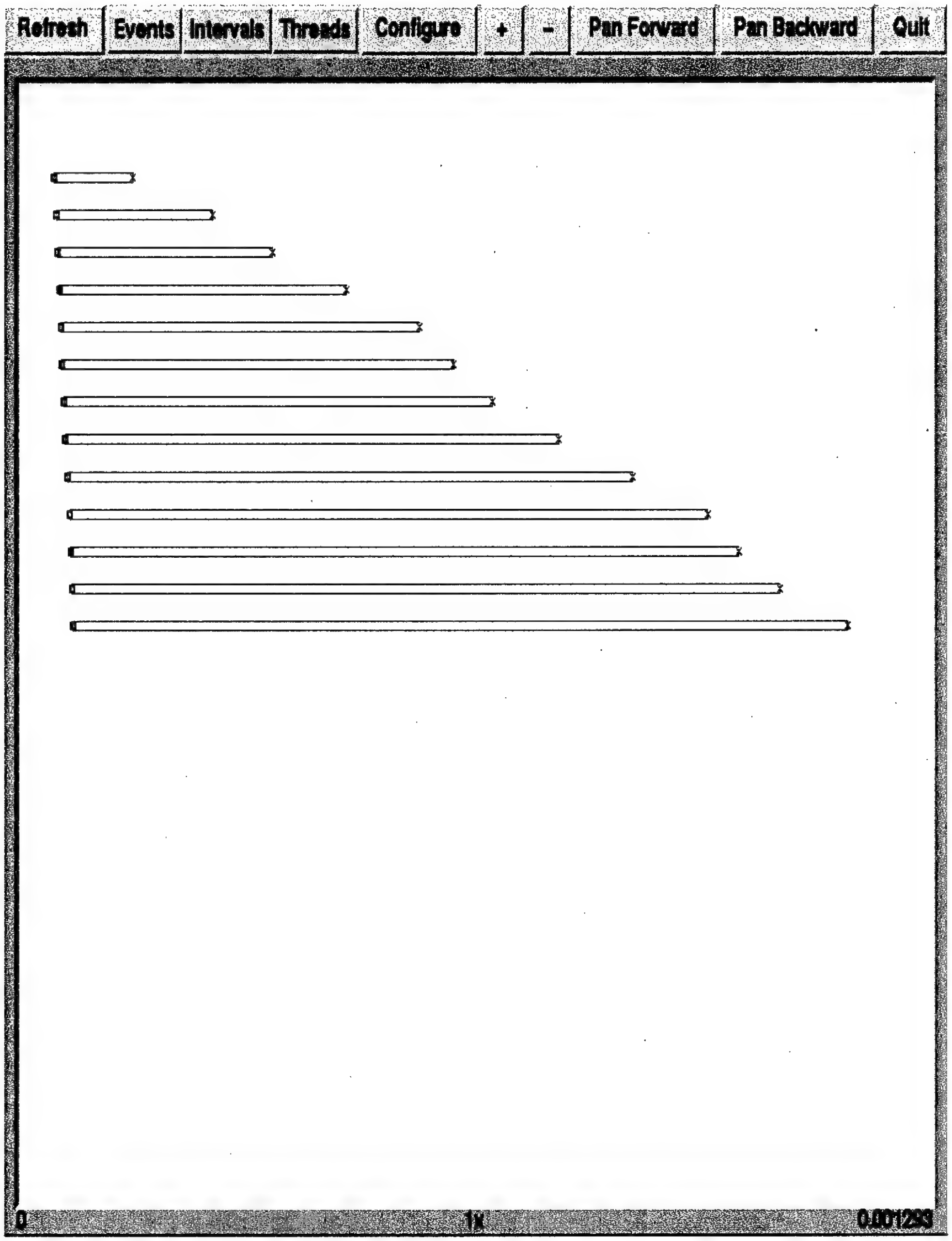

Figure 12. Plot of `StartMotorArrInt` intervals for experiment two. The horizontal bar indicates the interval between when the car is started and when it reaches the intersection. The time scale increases to the right. There are 13 cars in this test case; they are displayed from top to bottom.

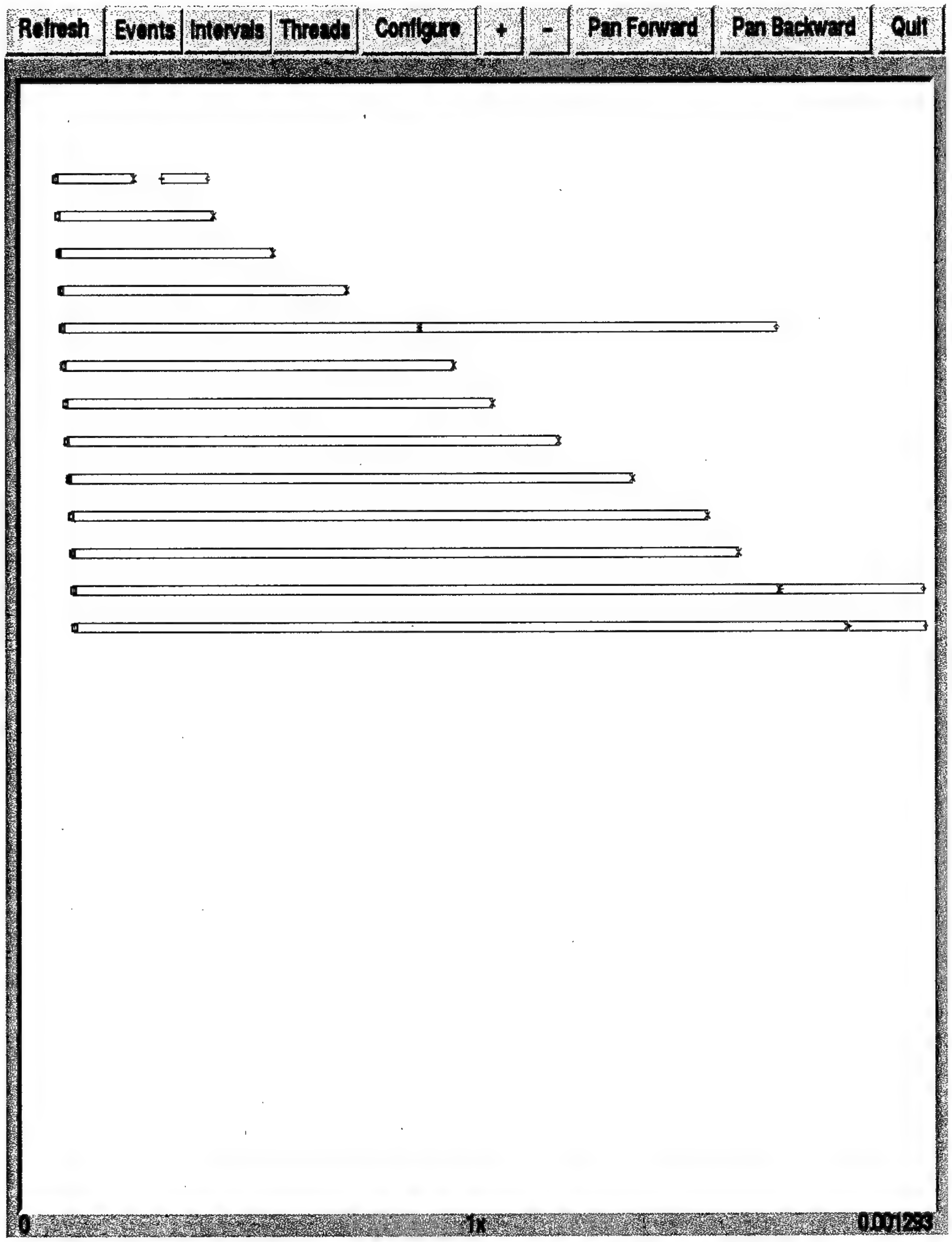

Figure 13. Plot of `StartMotorArrInt` and `StopAndGo` intervals for experiment two. The `StopAndGo` intervals are bound by `+` and `diamond` symbols. Only car 1, 5, 12, and 13 have a `StopAndGo` interval.

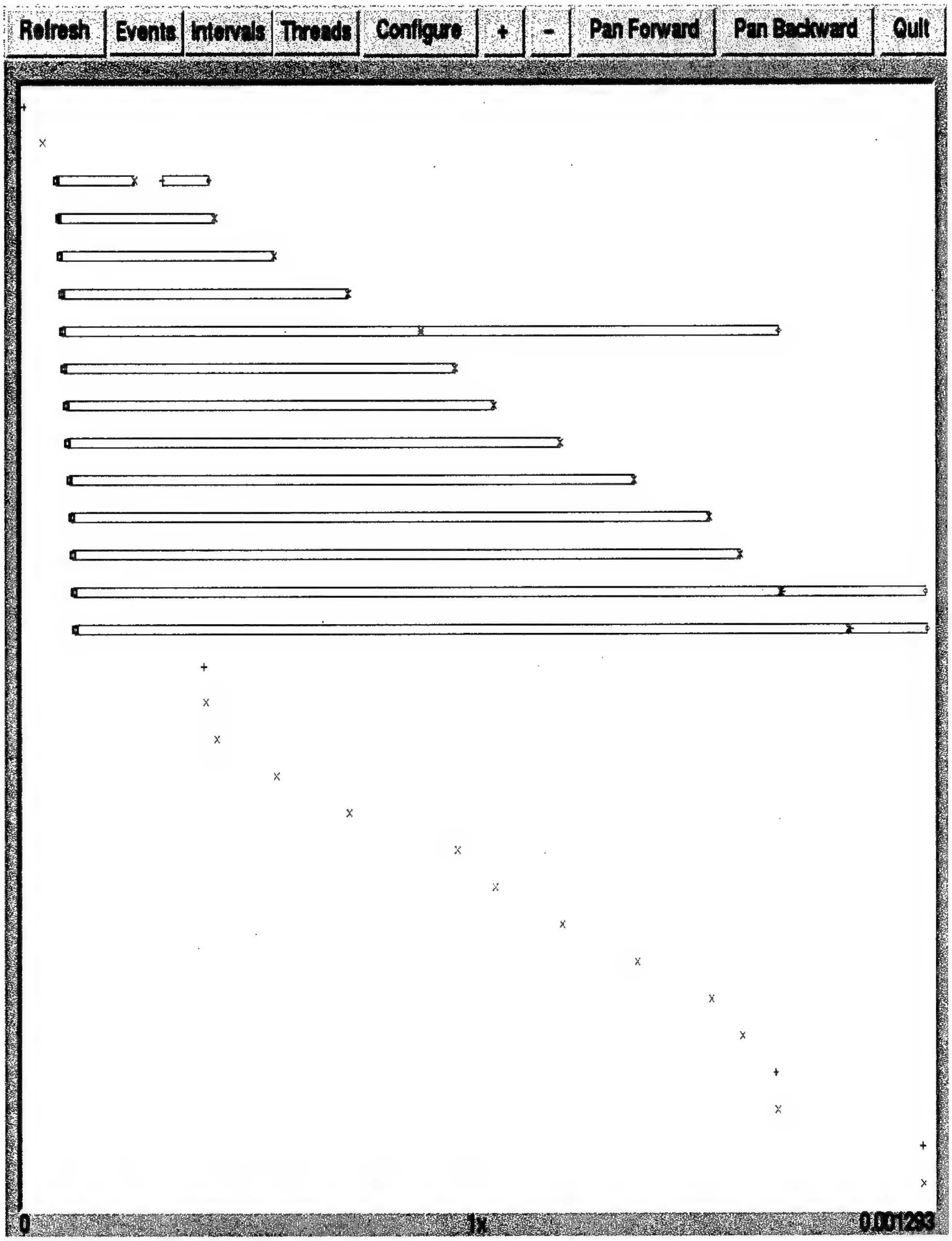

Figure 14. Plot of `LargeTimer` and `SmallTimer` events with `StartMotorArrInt` and `StopAndGo` intervals in experiment two. The symbol + and symbol **x** depict when the `LargeTimer` and `SmallTimer` objects are instantiated in the `StopLight` program. The first pair of timers is created (shown at the top of the plot) when the `Intersection` object is initialized, before any car objects are created.

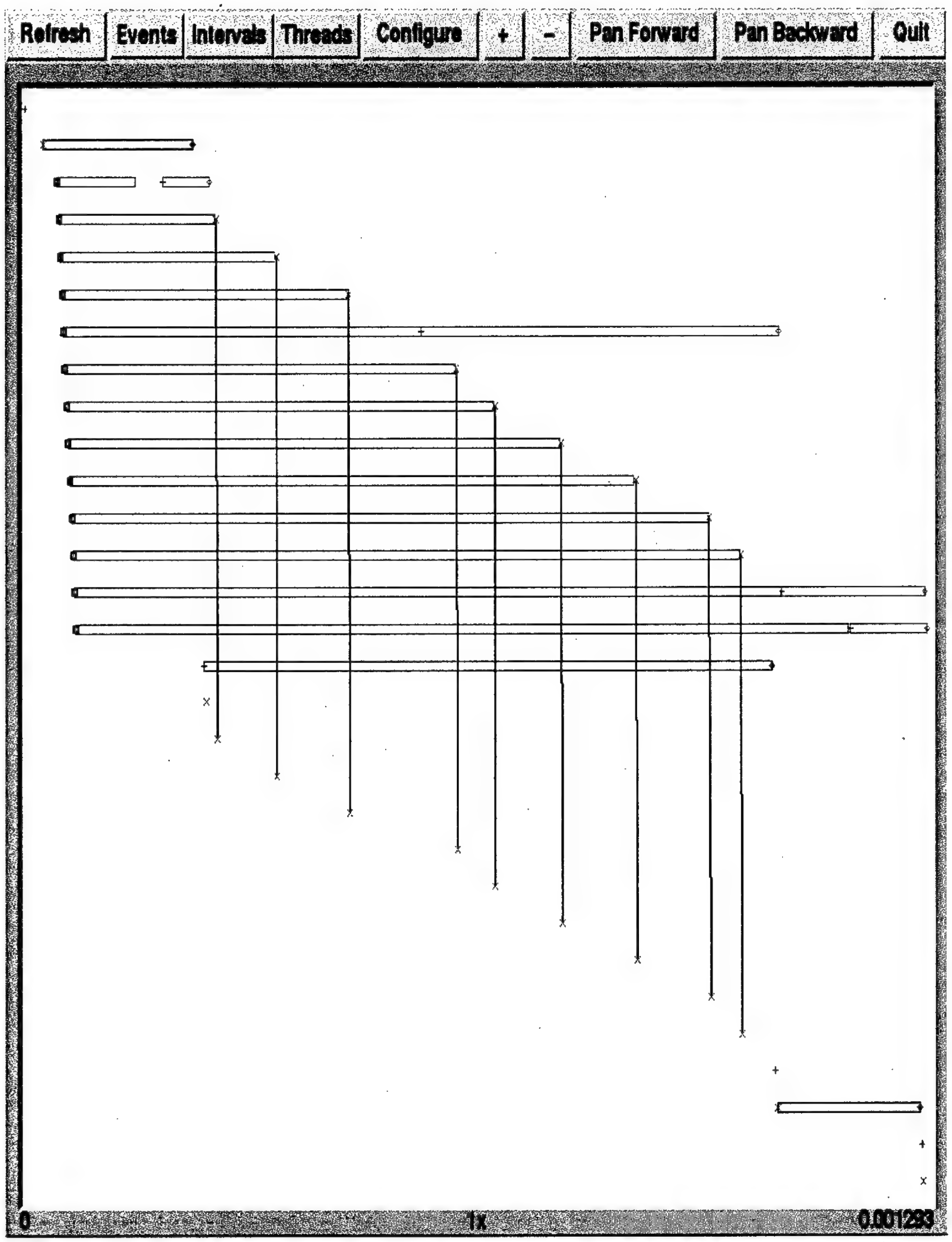

Figure 15. `NoWaitSmallTimer` interval is represented by the vertical line connection between the symbol **x** in the car thread and symbol **x** in the `SmallTimer`. There are 9 `NoWaitSmallTimer` intervals, originating from car 2, 3, 4, 6, 7, 8, 9, 10 and 11. These intervals represent the `SmallTimer` objects created by cars that did not have to stop at the intersection.

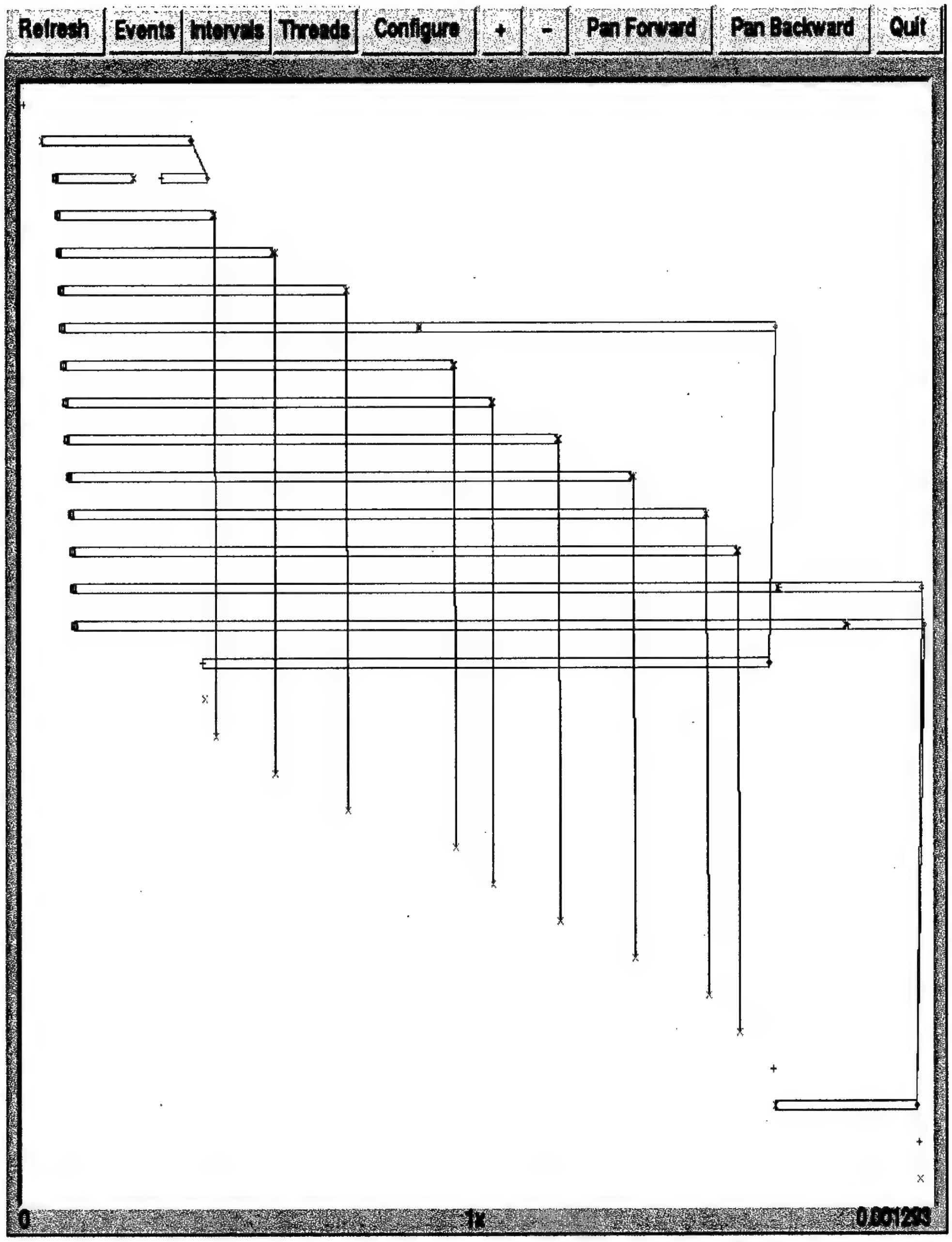

Figure 16. Interval `LtimerLtChanged` and `StimerLtChanged` are shown as the horizontal bar (between symbol `x` and symbol `diamond`). The interval `LtChangedAndGo` is shown as a vertical line from the right edge of the `LtimerLtChanged` or `StimerLtChanged` interval in the timer thread to the right edge of the `StopAndGo` interval in the car threads. These intervals represent the notification received by the car threads when the traffic light is changed by the timer threads.

the interval `LtimerLtChanged` that connects the car threads and the last `SmallTimer` thread.

4. Summary

In this chapter, we described a multi-threaded Java program that was used to demonstrate the working of the *Glimpse* Data Collection Facility. We then defined the event and interval specifications that are used to describe the behavior of this test program, in particular the interaction between threads in this application. By running the test program with two different inputs, we collected two experimental data sets. Using the visualization tool *gorge*, we were able to verify that event data collected from the experiments are consistent with the behavior of the application.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. APPLYING GLIMPSE TO AUTOMATICALLY PARALLELIZED FORTRAN PROGRAM

In this chapter, we describe our experiences porting *Glimpse* to monitor the performance of automatically parallelized FORTRAN programs. The FORTRAN programs that we expect *Glimpse* to be useful with are those that use lightweight processes to implement concurrent execution on systems with multiple processors. Although the FORTRAN parallel programming model and the language constructs that FORTRAN uses are quite different from the models and constructs in both Java and C++, we found that the basic approach used *Glimpse* is applicable to the performance analysis of parallel FORTRAN codes. In particular, the language-independent concepts of *events* and *intervals* proved quite useful.

In section one, the OpenMP tool for parallizing FORTRAN code is described. In section two, the modifications we made to *Glimpse's* Data Collection Facility to work with automatically parallelized FORTRAN programs are discussed. In section three, results from testing *Glimpse* with parallel FORTRAN programs are analyzed. In particular we see that different work-load scheduling algorithms have a large impact on the execution time of parallel programs and how *Glimpse* can be useful in understanding this impact. The test programs we used include a simple, loop-level parallel code and an operational weather forecast application. We conclude in section four by describing the experiences that we gained from modifying *Glimpse* to be used with automatically parallelized FORTRAN code and actually using it.

1. Parallelizing Programs using OpenMP

OpenMP [Ref. 18] is a tool from SGI that can be used by FORTRAN and C programmers to allow their code to take advantage of multiple processor environments. In many parallel programming approaches, such as those that use POSIX threads or Message Passing Interface (MPI), programmers must explicitly structure

the program by adding necessary function calls to the source code that handle synchronization, data movement, and other communication between parallel execution tasks. The OpenMP approach, on the other hand, relies on the compiler to handle such details. To set up a program for concurrent execution, the programmer inserts directives for the sections of the code that can safely be executed in parallel. A primary target for these directives is the loop constructs. The OpenMP-enabled compilers process the directives and generate calls to library functions. These library functions create lightweight processes at runtime and distribute loop iterations among them. [1]

The following example (in FORTRAN) illustrates the use of multiprocess directives:

```
      program mploop

      integer maxlen
      parameter(maxlen = 400)
      real*8 a(maxlen), b(maxlen), c(maxlen)
      integer len
      len=maxlen
C
C$DOACROSS SHARE(a,b,c,len),LOCAL(i),MP_SCHEDTYPE=SIMPLE
C$& CHUNK=25
C
      do i = 1, len
          a(i) = b(i) + c(i)
      end do
      stop
      end
```

[1]In this thesis we use the term lightweight process as it is used by SGI. SGI's definition is slightly different from these definitions typically found in textbooks and papers. In their definition, a lightweight process shares its address space with its parent process. In that respect, a lightweight process is similar to a thread. However, their lightweight process has 2 characteristics: (1) it carries a full set of the state information of a process, and (2) the dispatch of their lightweight processes is done in the kernel space. On the Silicon Graphics systems, a lightweight process is created by the **sproc()** system call.

The multiprocess directives are placed in the comment section of the code. They begin with a 'C' in column one and a '$' as the second character. The directives may be followed by optional clauses. If the compiler has multiprocessing turned off, these statements have no effect. This feature allows the identical source code to be compiled for a single processor system, e.g., for debugging purposes, without the multiprocess option. The example demonstrates the use of the `DOACROSS` directive. This directive indicates to the compiler to generate code that will execute multiple iterations of a `DO` loop in parallel. The `C$DOACROSS` directive applies only to the next executable statement that follows, which must be a `DO` loop. The `SHARE` and `LOCAL` clauses specify lists of variables used within the ensuing parallel region. The `LOCAL` clause specifies variables that are local to each parallel process. A variable is declared as LOCAL if its value does not depend on any other iteration of the loop and its value is used only within a single iteration (e.g., the loop-iteration variable `i` in this example). The `SHARE` clause specifies variables that are shared across all parallel processes. If a variable is declared as SHARE, all iterations of the loop use the same copy of the variable. A variable is declared as SHARE if it is only read (not written to) within the loop or if it is an array where each iteration of the loop uses a different element of the array [Ref. 19].

The `MP_SCHEDTYPE` and `CHUNK` clauses affect the way the compiler and run-time environment schedule work among the participating parallel processes in a loop. The default scheduling algorithm (e.g., `SIMPLE`) divides the iterations of the DO loop equally among all participating tasks. Other MP_SCHEDTYPE options are DYNAMIC, INTERLEAVE, and GUIDED SELF-SCHEDULING. These clauses do not affect the correctness of the loop, but can impact the performance of the critical loop. The effect of different work scheduling algorithms will be described in section 3 of this chapter.

The advantage of the compiler-directed approach to automatically parallelizing code is that it hides most of the low-level details associated with parallel program-

ming from the programmers. By adding a few simple directives and hints to the parallel regions of the source code, one can potentially speed up the execution of the application. Additionally, it is simple to change from one type of code partitioning to another. As discussed in section 3 of this chapter the type of scheduling desired can be application specific. Of course, performance bottlenecks will still occur for code parallized by this technique; in particular, in the distribution of workload among the participating tasks and in the interaction with the runtime environment. In the following section, we describe the modifications to *Glimpse* to allow it to monitor such potential performance bottlenecks.

2. Adapting Glimpse to Monitor Multiprocessing FORTRAN Code

Following the approach used for adapting *Graze* to the Java environment, the FORTRAN version of *Glimpse* will use the event and interval specification (Appendix B) and will provide the capability to translate event specifications into event logging subroutines that can be called by the FORTRAN application. The design objectives for the Java implementation (Chapter IV) were also considered in the FORTRAN implementation, that is, (i) maintaining the log file structure and naming convention for compatibility with existing visualization tools, (ii) keeping the number of function calls needed to capture event data to a minimum, and (iii) separating the part of data logging code that is dependent on the event specification from that which is independent in order to improve modularity.

Based on these considerations, we implemented a front-end program, **builder_fortran** and a new utility library, **glimpse_io.a**. The **builder_fortran** program parses the event specifications and generates FORTRAN subroutines. The **glimpse_io.a** library contains utility functions, such as those needed for opening log files, writing to log files, and obtaining timestamp information. The library maintains a list of log file records at runtime, one for each process in the parallel FORTRAN application. A log file record is a C language `struct` that contains information such

as the name of the log file and the file pointer to the log file, as well as a pointer to an internal buffer in which to store the event data. During the execution of parallel FORTRAN programs, the event logging subroutine looks up the appropriate log file record assigned to the given execution process and writes the event data to that buffer. When the event buffer is full, the library writes out the data to the appropriate log file. Unlike the Java programs where some threads might be short lived, the participating processes in the parallel FORTRAN code exists for the duration of the program. [2] Consequently, the method for looking up the appropriate log file record for a given process is straightforward. For example, for a FORTRAN program running with 4 processes, the first process is associated with the element 0 of the log file record array and data from that process are written to 'log.0.'

The utility functions in **glimpse_io.a** library are implemented in the C language. To facilitate implementation of FORTRAN data logging subroutines, the FORTRAN interface to the library is provided by the following functions: `GLIMPSE_INITIALIZE`, `GLIMPSE_CLOSE`, `LOG_EVENT`, and `LOG_EVENT_ATTR`. `GLIMPSE_INITIALIZE` function creates the necessary log file records and initializes the log files. The `GLIMPSE_CLOSE` function flushes out any event data in the memory buffer before closing the log files. The `LOG_EVENT` function writes out the event id and its timestamp to the specified buffer and the `LOG_EVENT_ATTR` function writes out the event attribute. Consider the following event.

```
event write_data(size)
```

The **builder_fortran** program generates the following FORTRAN subroutines:

[2]For example, on the Silicon Graphics systems, the main process creates a group of lightweight processes at the start of the program. When the main thread of execution reaches a parallel region in the code, i.e., the `Do` loop, these lightweight processes are unblocked and become active participants in the execution of the `Do` loop. After leaving the parallel region, the lightweight processes become blocked again until the main execution thread reaches another parallel region. The number of lightweight process, a user controllable parameter, is set prior to the program execution. It cannot be changed once the program execution starts.

```fortran
c
c  File: fglimpse.f
c
c  Subroutine to intialize the glimpse io package
c
      subroutine glimpse_init()
         integer num_of_proc(1)
         num_of_proc(1)=mp_numthreads()
         call GLIMPSE_INITIALIZE(num_of_proc)
      return
      end
c
c  Subroutine to close the glimpse io package
c
      subroutine glimpse_terminate()
         call GLIMPSE_CLOSE()
      return
      end
c
c  Subroutine to log event
c
      subroutine write_data(size)
         integer tid(1)
         integer event_id(1)
         integer event_attr(1)
         tid(1)=mp_my_threadnum()
         event_id(1)=0
         call LOG_EVENT(tid,event_id)
         event_attr(1)=size
         call LOG_EVENT_ATTR(tid,size)
      return
      end
```

The FORTRAN subroutine, `glimpse_init()`, initializes the Data Collection Facility by calling the C-function `GLIMPSE_INITIALIZE` to create the necessary log file records and initialize the log files. The function `mp_numthreads()` returns the number of processes that will be used to execute the parallel FORTRAN code. This number is passed to GLIMPSE_INITIALIZE during the initialization. The subroutine `glimpse_terminate` calls the C-function GLIMPSE_CLOSE to flush out any event

data in the memory buffer to files and shutdown the Data Collection Facility. The FORTRAN subroutine for logging the `write_data` event calls the utility functions LOG_EVENT and LOG_EVENT_ATTR to record the event id, timestamps, and event attribute information. The function `mp_my_threadnum()` is called inside the event logging subroutine to identify the process in which each particular event occurs.

The `mp_my_threadnum()` and `mp_numthreads()` are functions in the multi-tasking library provided by the system vendor. Compilers that process the multi-process directives use functions from this library to implement the parallel execution of FORTRAN programs. By isolating these system specific calls to the FORTRAN component of the Data Collection Facility, only the **builder_fortran** program needs to be modified to generate the appropriate calls for another vendor library if we move to another platform. The utility functions in the **glimpse_io.a** library are platform independent.

3. Results from Monitoring Parallel FORTRAN programs

In this section we describe our experiences monitoring automatically parallelized FORTRAN programs using the modified *Glimpse* software. We concentrate on how the workload (e.g., the loop iterations) is divided among the participating processes in the parallel region of the code. Before discussion the results, we will first describe the algorithms that the runtime multiprocess library uses to divide up the loop iterations. We recall from the discussion in section one that there is a 'MP_SCHEDTYPE' clause in the compiler directive that can be set, by the user, to guide the scheduling in parallel region. The allowable values of 'MP_SCHEDTYPE' and their meanings are list below [Ref. 18]:

Workload Schedule	Meaning
`SIMPLE`	Each process executes N/P consecutive iterations starting at Q * (N/P) iterations. The first process to finish takes the remaining chunk, if any.
`DYNAMIC`	Each process executes C iterations of the loop, starting with the next undone chunk unit, returning for another chunk until none are left.
`INTERLEAVE`	Each process executes C iterations at (C * Q) + base, (C * 2Q) + base, (C * 3Q) + base, etc. The base = 1, 1 + (C * P), 1 + (C * 2P), etc.
`GSS`	Guided Self-Scheduling, each process executes chunks of decreasing size, first N/2P, then N/4P, etc

The variables used in the Table are as follows:

`N`-Number of iterations in the loop, determined from the source or at run time.

`P`-Number of participating lightweight processes, set by the default or by the user via an environment variable.

`Q`-Unique identification of a lightweight process, in the range from 0 to P-1.

`C`-"Chunk" size, set by compiler directive or by environment variable.

The `SIMPLE` method divides the loop iterations among processes by dividing them into contiguous pieces and assigning one piece to each process. For example, if N is 500, and the number of processes P is 4, then process 0 is assigned loop iterations 1 to 125, process 1 is assigned loop iterations 126 to 250, etc.

In the `Dynamic` method, the iterations are divided into pieces. The size of a piece is specified with the CHUNK clause. As each process finishes a piece, it grabs the next available piece. For example, if N is 500, P is 4, and CHUNK size is 25, then a possible scenario for process 0 can have it works on iteration 1-25, 101-125, 151-175, 201-225, 301-325, 401-425, 476-500, etc. When a process runs on a CPU

without much competition for CPU cycles, it may process more than an equal share of chunks.

The `Interleave` method also breaks the iterations into pieces of size specified by the CHUNK variable. Execution of those pieces is interleaved among the processes. For example, for P=4, N=500, C=25, process 0 would work on iterations 1-25, 101-125, 201-225, 301-325, 401-425; and process 1 works on iteration 26-50, 126-150, 226-250, etc. Although the number of iterations executed by each process is the same for SIMPLE and INTERLEAVE, INTERLEAVE can outperform SIMPLE when the amount of work to be done in an iteration is a function of iteration number.

In the `GSS` method, the size of pieces varies depending on the number of iterations remaining. In the beginning, larger pieces are parceled out, and as the work nears completion, smaller pieces are parceled out.

We use the example program (see below) from section one to demonstrate the value of the modified *Glimpse* with automatically parallelized FORTRAN code. Two data logging calls, `event_x()` and `event_y()`, denote the entry and exit events for each iterations inside the parallel region. A call to glimpse_init() initializes the Data Collection Facility, and a call to glimpse_terminate() flushes event data in the memory buffers to the log files.

```
      program mploop

      integer maxlen
      parameter(maxlen = 400)
      real*8 a(maxlen), b(maxlen), c(maxlen)
      integer len
      len=maxlen
C
C  Initialize glimpse Data Collection Facility
C
       call glimpse_init()
C
C$DOACROSS SHARE(a,b,c,len),LOCAL(i),MP_SCHEDTYPE=SIMPLE
C$& CHUNK=25
C   MP_SCHEDTYPE=DYNAMIC
```

```
C   MP_SCHEDTYPE=INTERLEAVE
C   MP_SCHEDTYPE=GSS
C
      do i = 1, len
          call event_x()
          a(i) = b(i) + c(i)
          call event_y()
      end do
C
C   Close glimpse Data Collection Facility
C
      call glimpse_terminate()
      stop
      end
```

To evaluate the effect of various workload schedules, we ran the example program multiple times with each schedule mode under different load condition. The number of processes, P, is set to 4, the number of iterations, N, is 400, and the chunk size, C, is 25, for all runs. The example program was run on a 6-processor Silicon Graphics R12000 platform, under various load conditions. The load of the system during the execution was recorded using the UNIX 'w' command. The load on the system gives an indication of the average number of jobs in the CPU queues. For the purpose of this discussion, we consider a load value of less than 1 as low, a value between 1 and 3 as medium, and a value of 4 or greater as high.

Figures 17 and 20 show representative results for the example program running with the SIMPLE workload schedule under a low load condition. Each participating process grabs an equal share of workload and they all take about the same amount of time to complete their portion of the workload. This is shown by all 4 processes reaching the event count of 100 at about the same time (Figure 17), and none of the participating processes experience any delay (Figure 18). In some instances, one of the processes is temporarily blocked. This is shown by the horizontal line in Figure 19 and the corresponding gap in the event sequences in Figure 20. The temporarily blocked processes took longer to finish. Because of the synchronization barrier at the

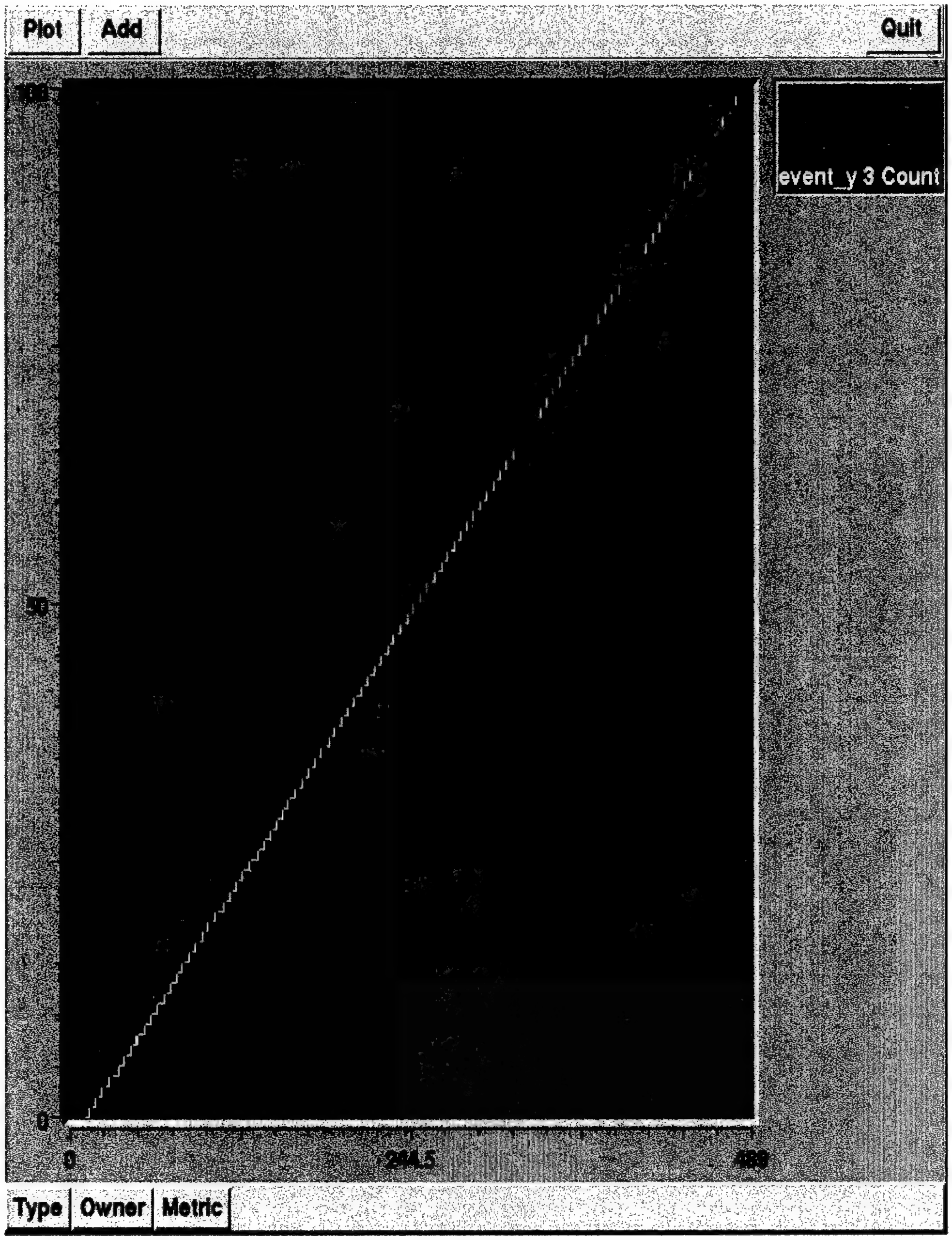

Figure 17. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using the SIMPLE schedule type under a low system load condition. Each event_y denotes the completion of a single loop iteration. The Time increases to the right and the number of counts is indicated on the vertical axis.

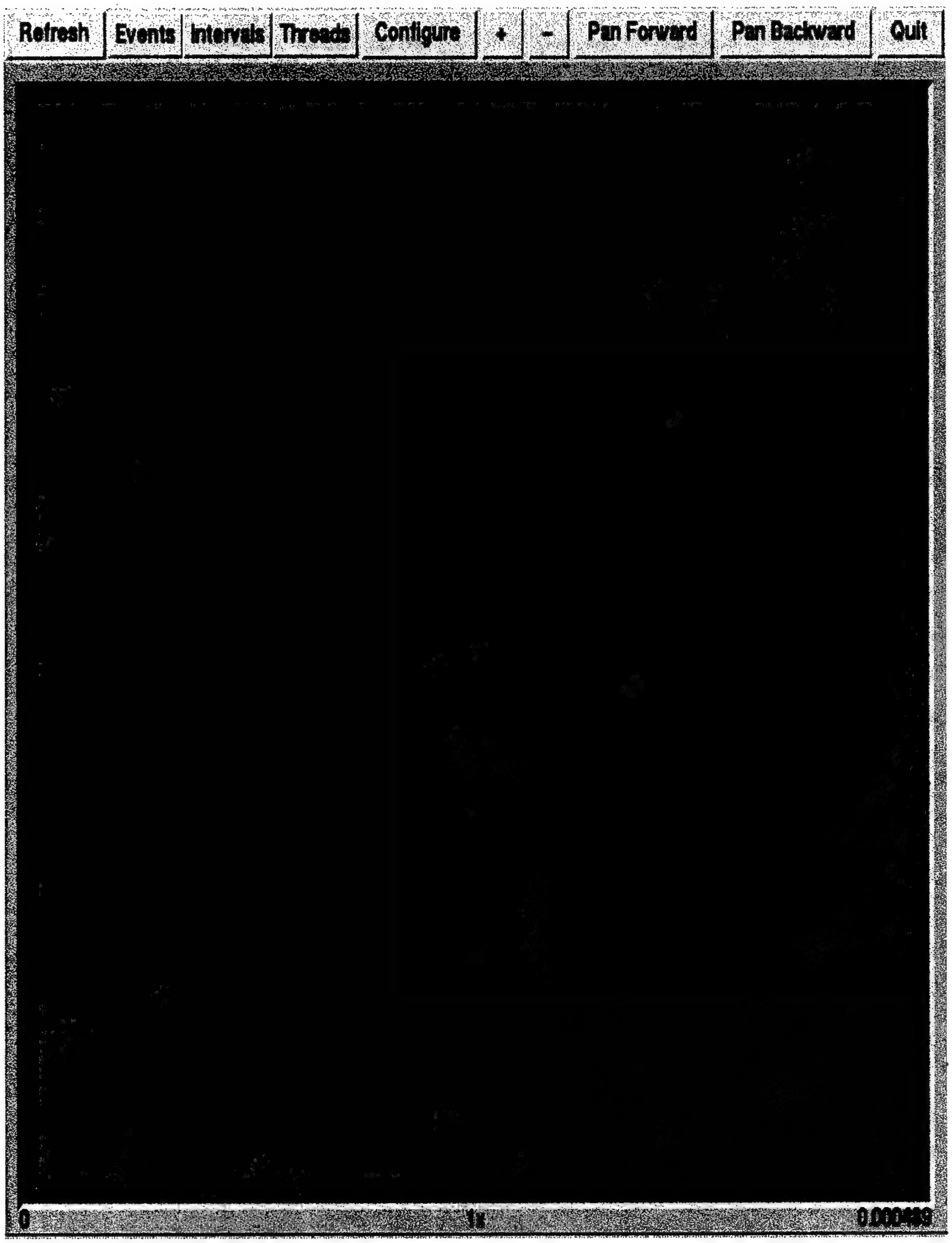

Figure 18. Plots of occurrences of event_y for 4 lightweight processes using the SIMPLE schedule type under a low system load condition (from the same data as in Figure 17). Each event_y denotes the completion of a single loop iteration. The Time increases to the right.

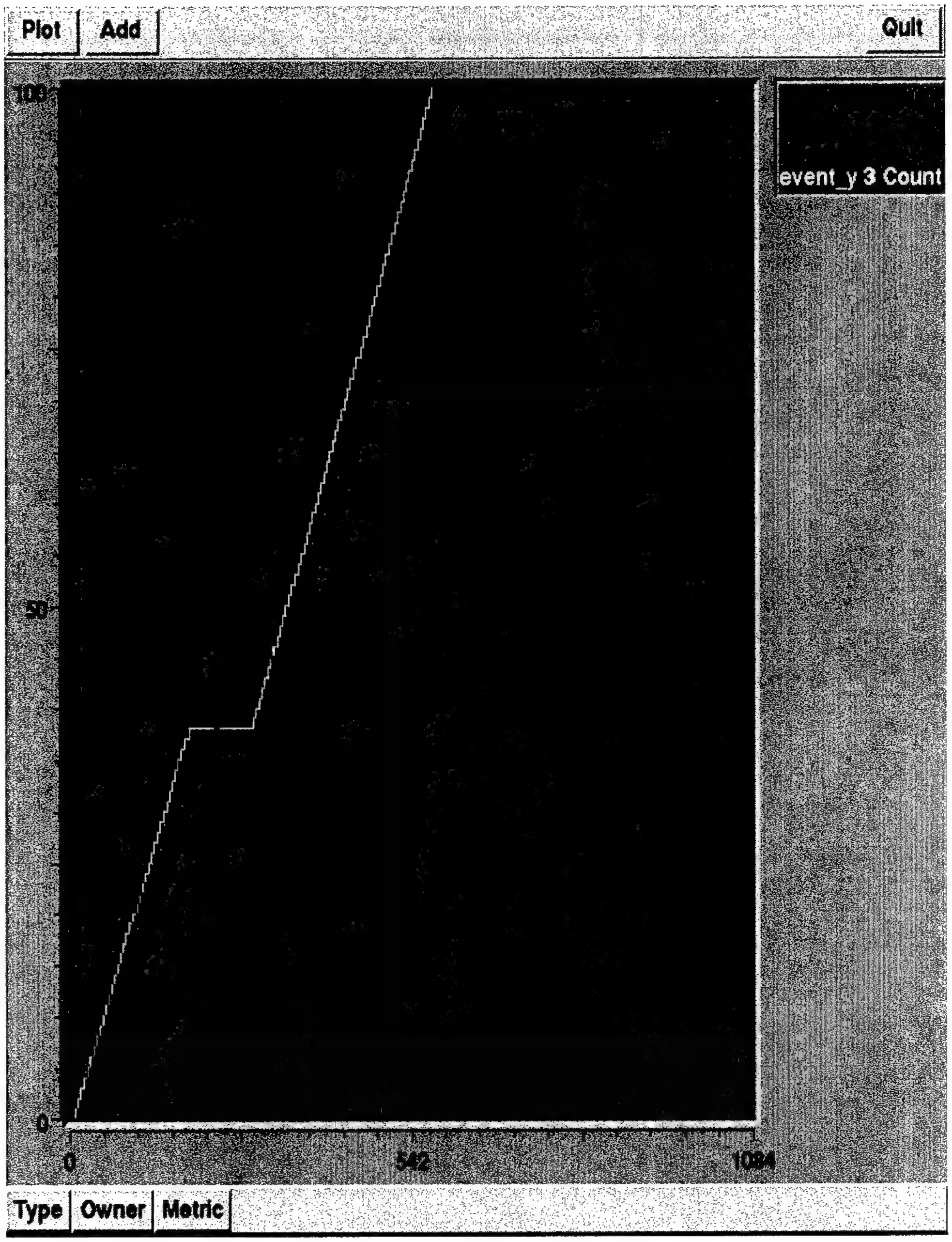

Figure 19. Plots of the counts of event_Y as a function of time. The plots are for 4 lightweight processes using the SIMPLE schedule type under a low system load condition. The horizontal line near count value 45 shows that the count value for that particular process is not increasing with time. This is an indication that the process is blocked during the duration that count value is not increasing. The Time increases to the right and the number of counts is indicated on the vertical axis.

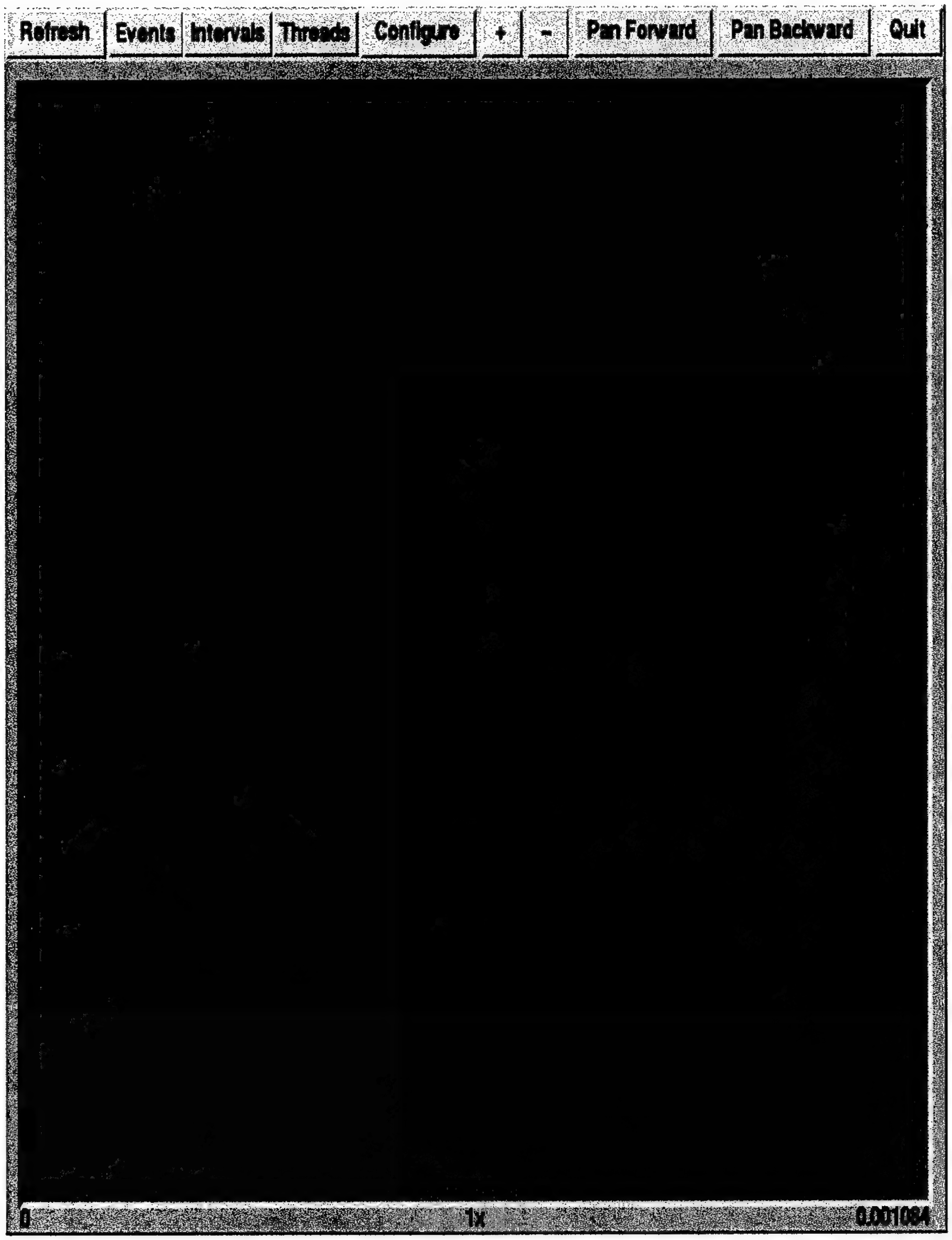

Figure 20. Plots of the event_y for 4 lightweight processes using the SIMPLE schedule type under a low system load condition (from the same data as in Figure 19). Each event_y denotes the completion of a single loop iteration. The large gap for process 0 (at the top of the graph) corresponds to the horizontal line of Figure 19. The gap indicates that the corresponding process is blocked during that period of time.

end of the loop, the completed processes must wait until the last process finishes. As a result, the overall execution time is longer in cases where blocking occurs in at least one process.

We next look at the example program running with the DYNAMIC workload schedule under low to medium load conditions. Figure 21 and 22 show results under a low load condition. The pattern is very similar to the SIMPLE schedule type in that each participating process grabs an equal share of the workload and they all require approximately the same amount of time to complete their portion of the workload. However, runs using the DYNAMIC schedule mode seem to take slightly longer time to complete than runs using the SIMPLE schedule mode under similar load conditions. This is probably due to the higher overheads associated with dynamic scheduling, given that the loop calculation time is very small.

Next results from running under a medium load condition with DYNAMIC scheduling are shown in Figure 23 and 24. Here, the number of iterations is not equally distributed across the processes. Process 3 was temporarily blocked, so it completes fewer iterations than the other processes. All of the processes finished at about the same time. As would be expected, the time it takes to complete the loop using DYNAMIC scheduling type under a low to medium load condition is, on average, slightly longer than the SIMPLE schedule type under a low load condition.

Next, we examine results using the GSS and INTERLEAVE scheduling policies. Under a low to medium load condition, the example program running with a GSS scheduling policy (Figures 25 and 25) exhibits similar behavior to that of the DYNAMIC scheduling policy. The workload across the processes is not equally distributed as in the DYNAMIC case, when a process finishes its chunk of workload, it grabs the next available chunk, so some processes perform more iterations than others. For the INTERLEAVE schedule type, the workload distribution across all of process is the same under our example (we chose the total number of loop iterations and chunk size such that each process gets 4 chunks, each of size 25). The average

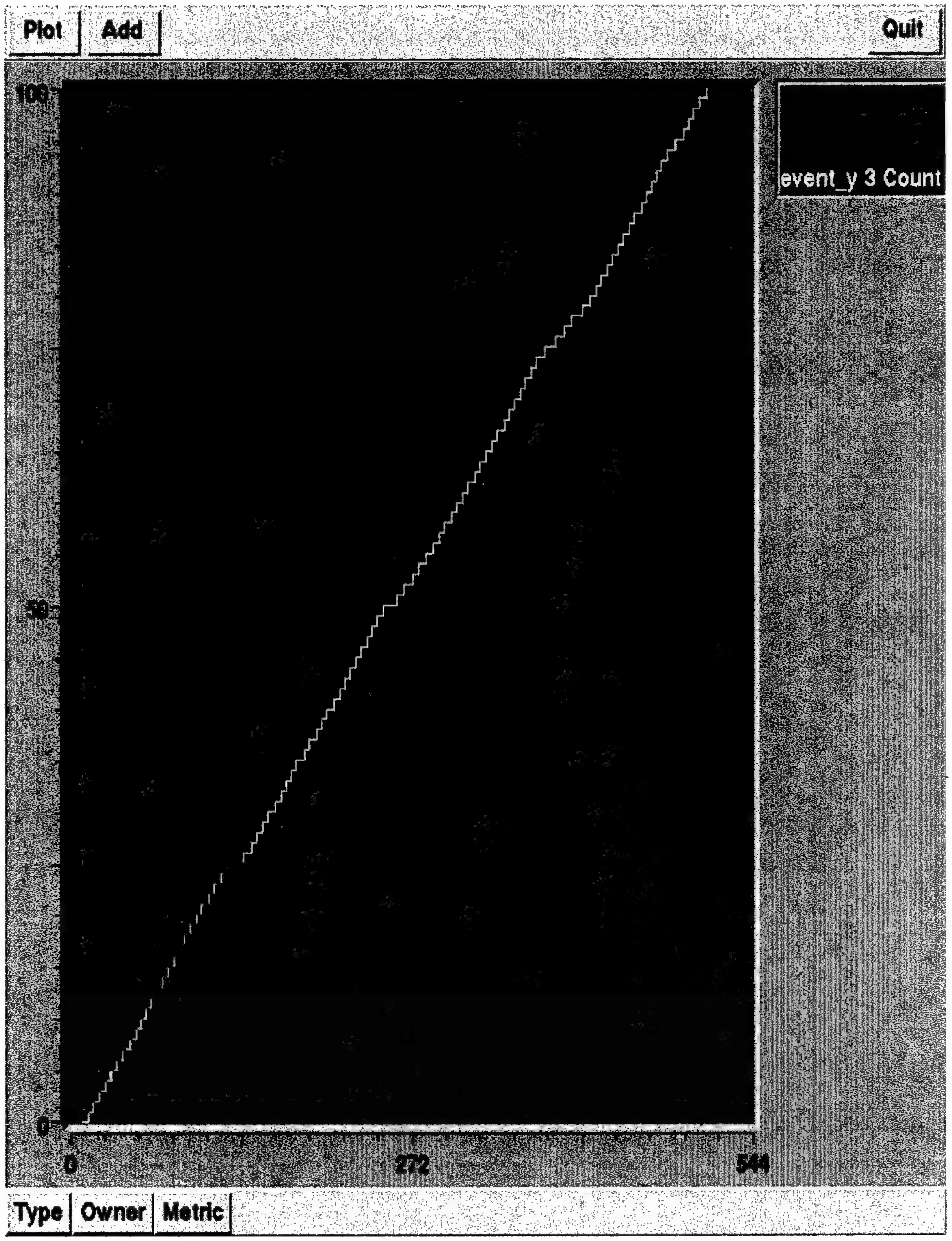

Figure 21. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using DYNAMIC scheduling under a low system load condition. Each event_y denotes the completion of a single loop iteration. The Time increases to the right and the number of counts is indicated on the vertical axis.

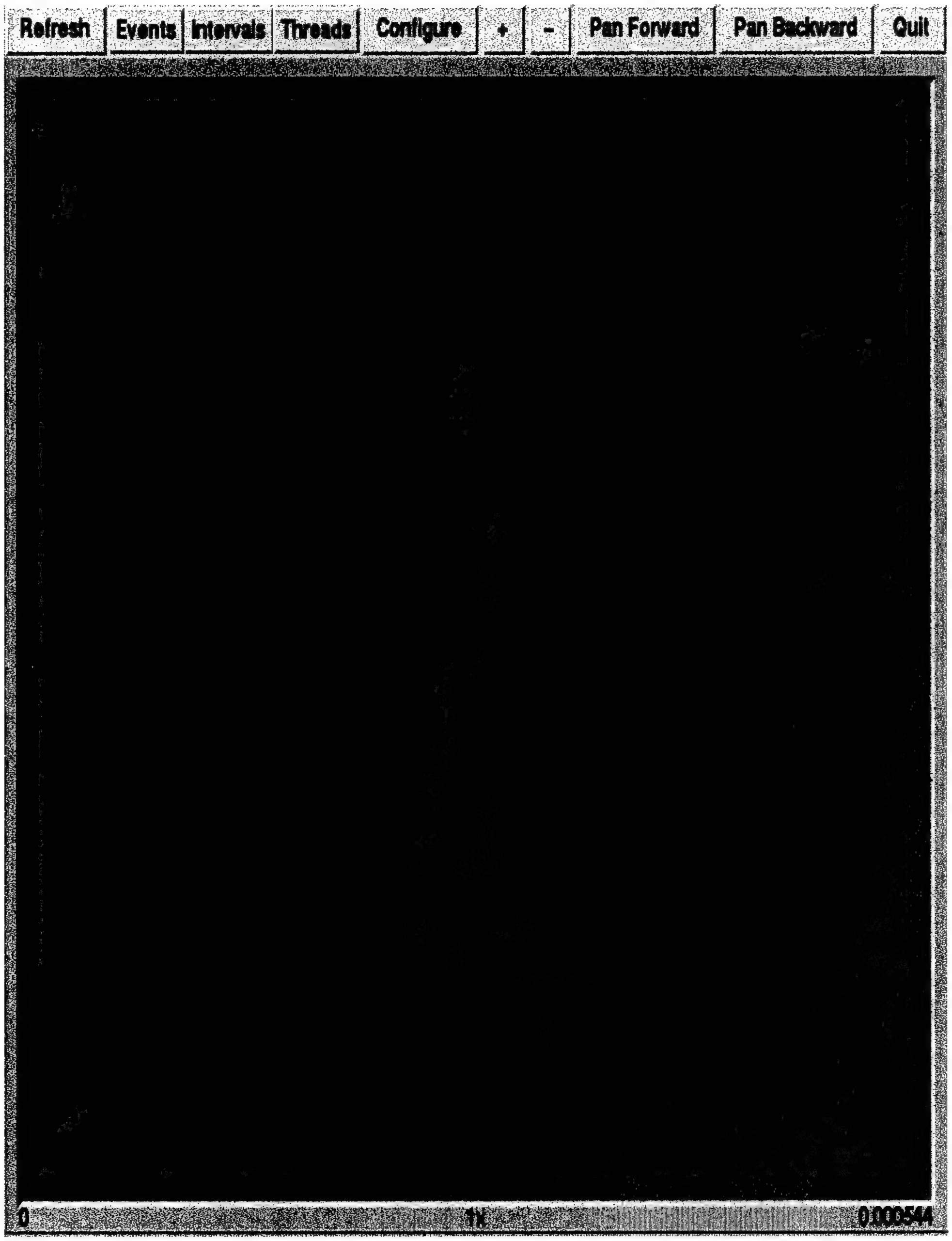

Figure 22. Plots of event_y for 4 lightweight processes using DYNAMIC scheduling under a low system load condition (from the same data as in Figure 21). Each event_y denotes the completion of a single loop iteration. The Time increases to the right.

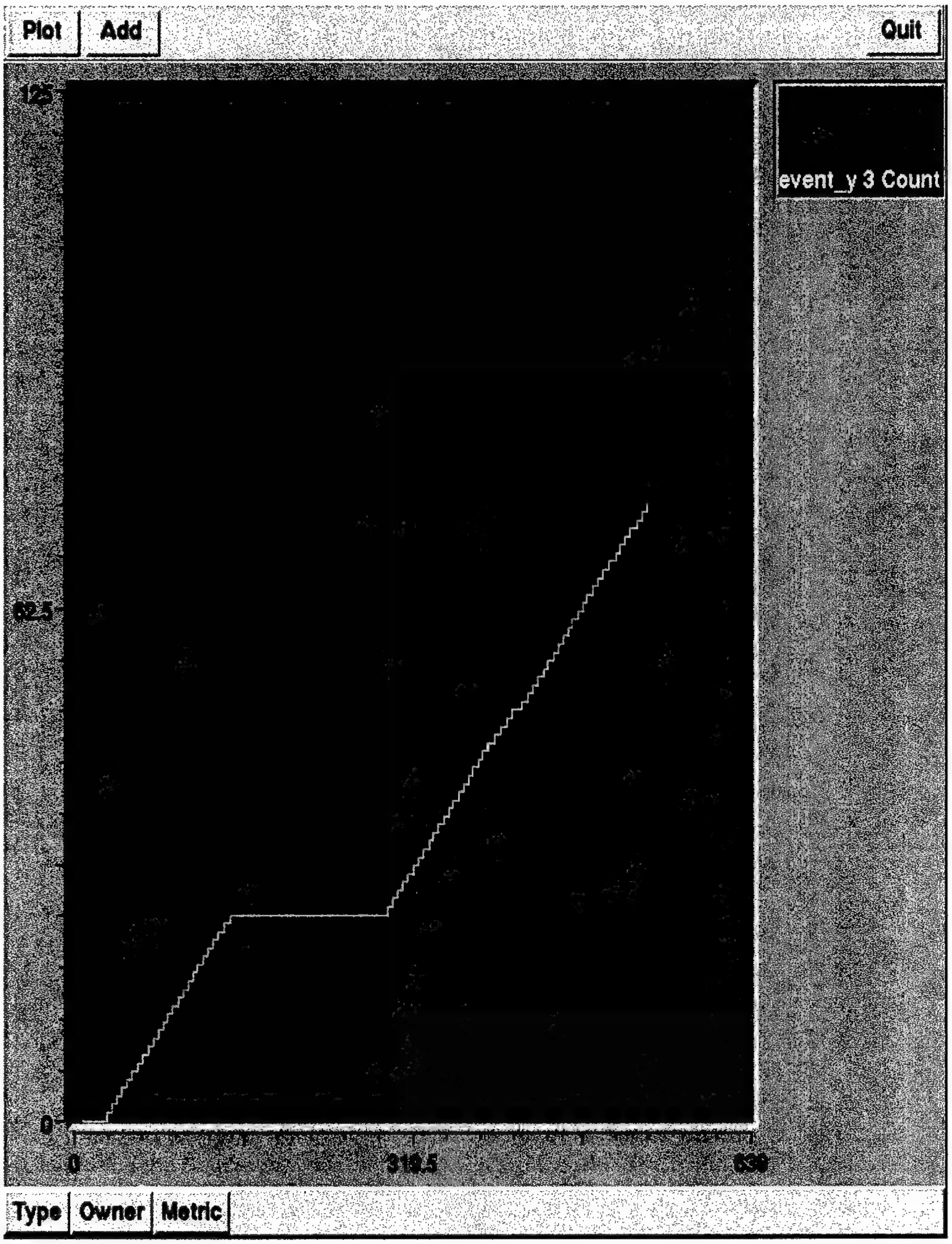

Figure 23. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using DYNAMIC scheduling under a medium system load condition. The horizontal line indicates that one of the processes is temporarily blocked during the execution in the Do loop. The Time increases to the right and the number of counts is indicated on the vertical axis.

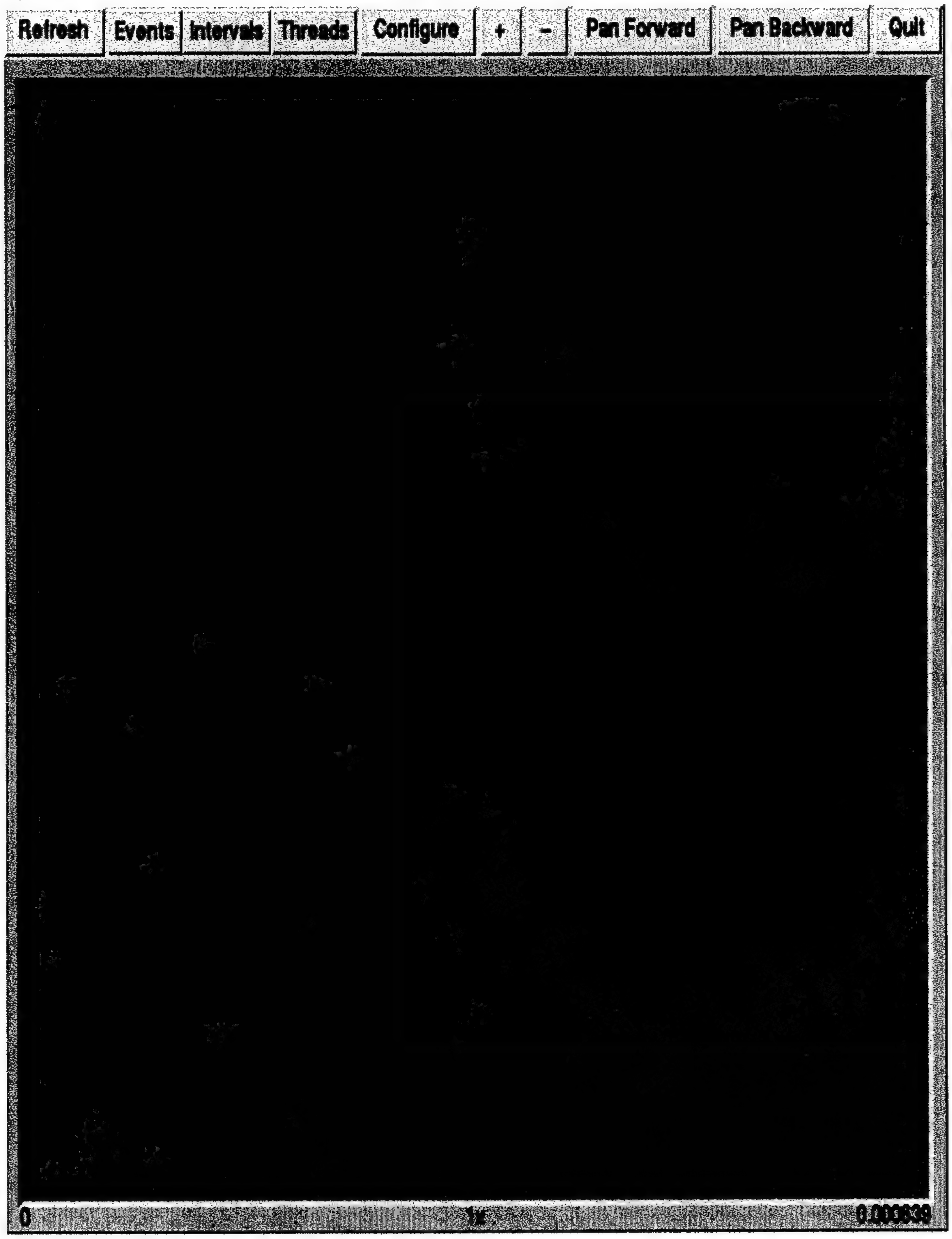

Figure 24. Plots of event_Y for 4 lightweight processes using DYNAMIC scheduling under a medium system load condition (from the same data as in Figure 23). Each event_y denotes the completion of single loop iteration. Process 0 (at the top of the graph) has the most number of event_y, i.e., it executes more iterations of the loop than other processes. Process 3 was blocked for a period of time (as shown by the gap between the event sequences). Consequently, it completes the least number of iterations of the loop.

time to finish the loop is about the same as with the SIMPLE schedule type. This result is expected because the purpose of interleaving iterations across the loop is to improve the access to shared arrays via cache optimization. Since our example uses small arrays (three 400 element arrays of double precision values), the cache optimization is not an issue.

We repeated the experiment with different schedule types by running the same example program under a high load condition. Figures 27 and 28 show the result for the SIMPLE schedule type under a high load condition. The execution time is significantly longer than under the low system load condition. In Figure 28 three of the processes have completed their work, but are forced to wait for the remaining process to finish. The waiting time accounts for most of the execution time for this program to complete.

Next, we look at the DYNAMIC schedule type under a high load condition (Figures 29 and 30). The plots show that although we request 4 processes, only 3 processes participate in the parallel region due to the high system load. The workload is not equally distributed among the 3 processes and all of them finish about the same time. The time to complete the loop for the example program using the DYNAMIC scheduling policy under the high load condition is longer than under the low load condition. However, it does not suffer from the significant performance degradation as experienced by the example program using a SIMPLE scheduling policy under a high load condition.

Next, we examine the results for GSS and INTERLEAVE scheduling types under a high load condition. The example program running with a GSS scheduling policy under high load condition (Figure 31 and 32) exhibits similar characteristics to that of running with the DYNAMIC scheduling policy. Like the previous case, although 4 processes were requested, only 3 processes participated in the parallel region due to high system load. The workload is not equally distributed among the 3 processes, but they all finish about the same time. The performance of the example

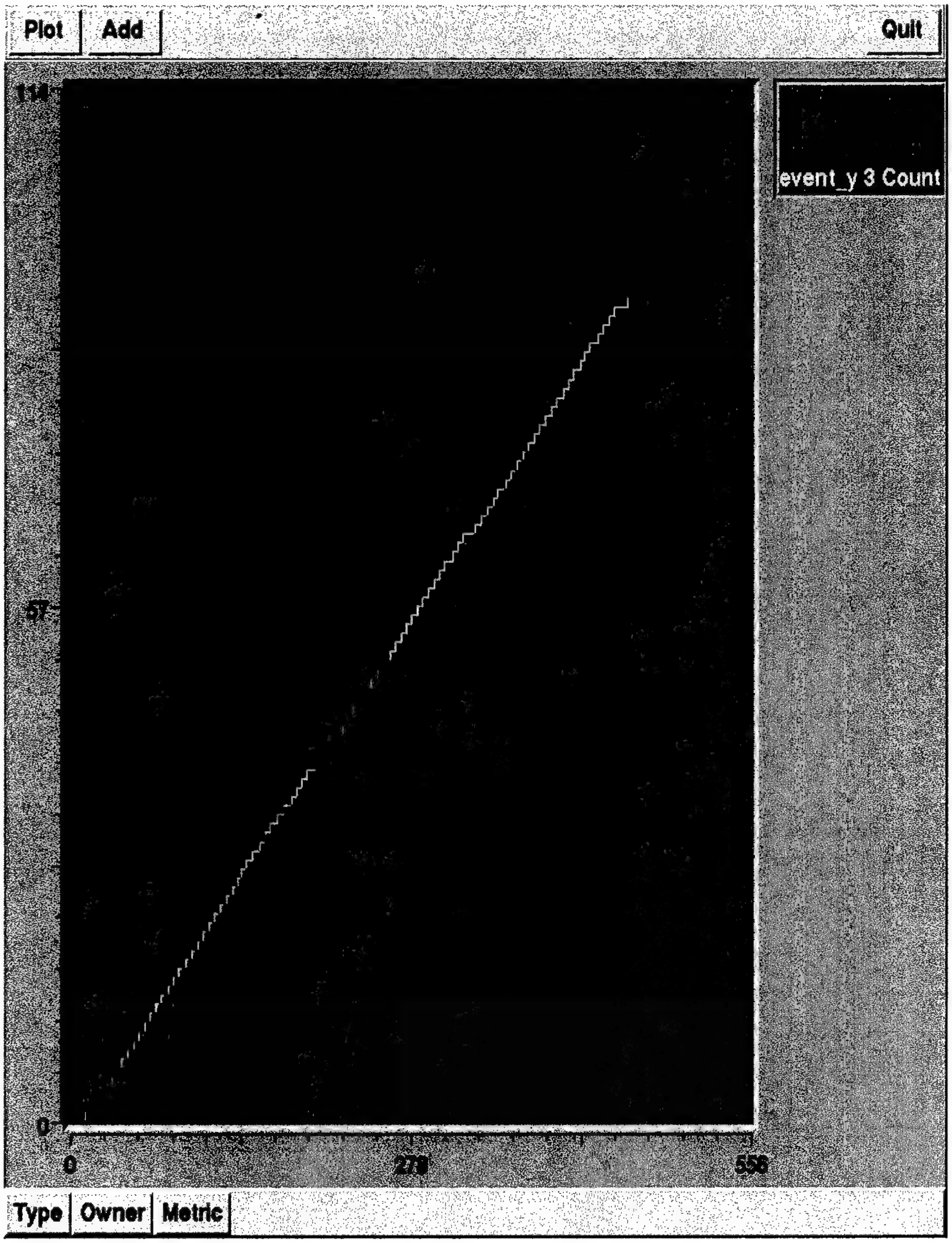

Figure 25. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using the GSS scheduling under a low system load condition. Each event_y denotes the completion of a single loop iteration. The Time increases to the right and the number of counts is indicated on the vertical axis.

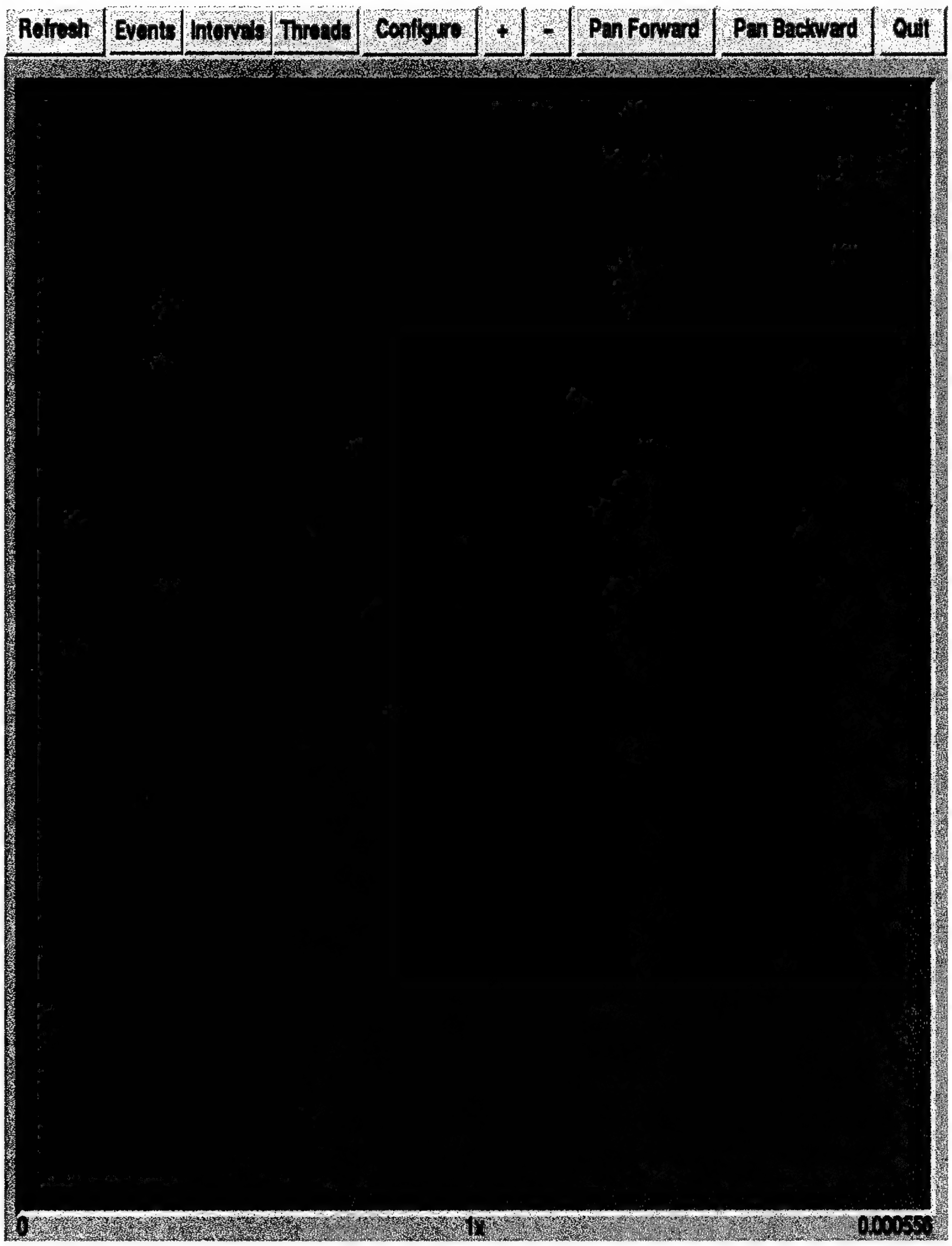

Figure 26. Plots of event_y for 4 lightweight processes using GSS scheduling under a low system load condition (from the same data as in Figure 25). Each event_y denotes the completion of a single loop iteration. The Time increases to the right.

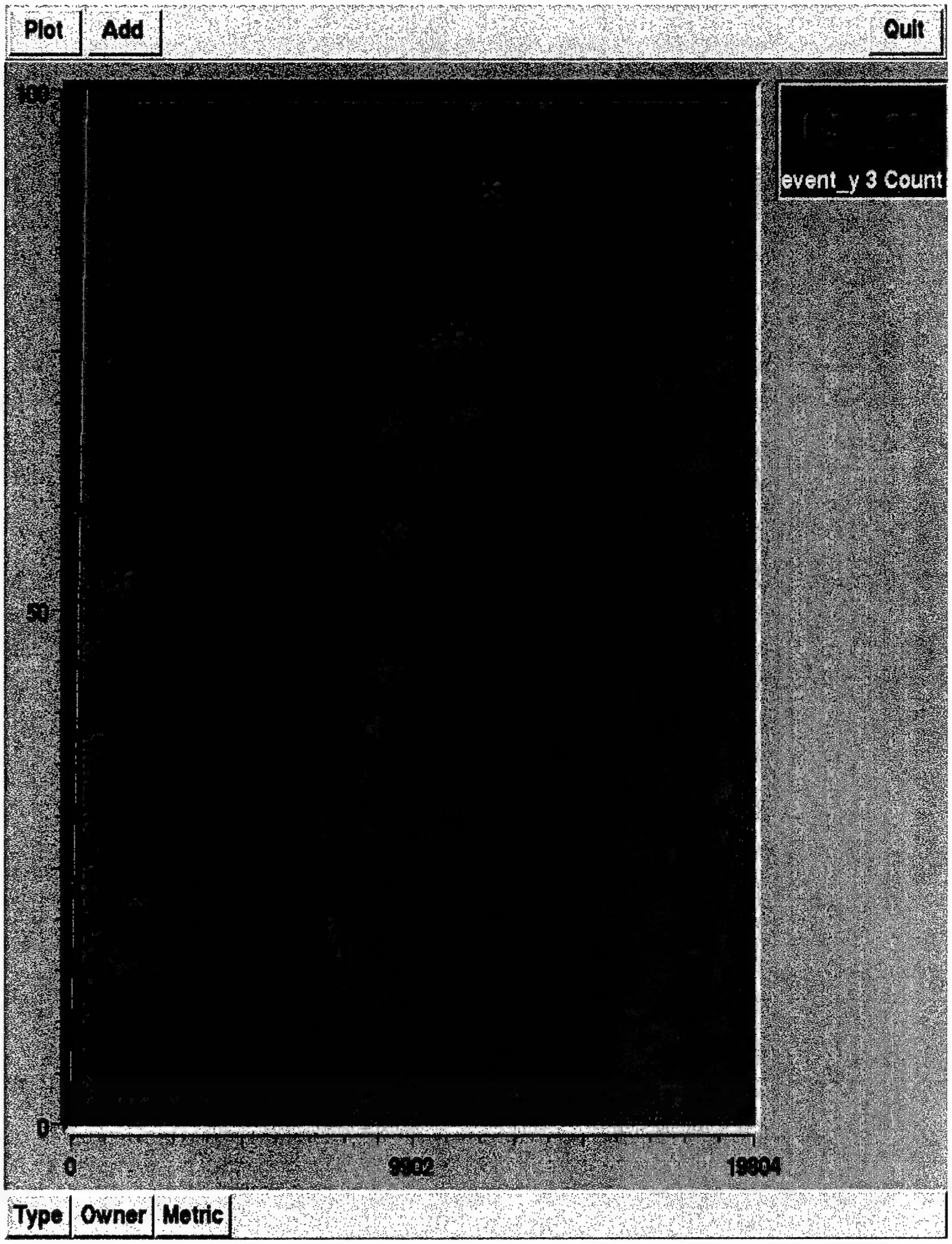

Figure 27. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using the SIMPLE schedule type under a high system load condition. Each event_y denotes the completion of a single loop iteration. The execution time of this case is significantly longer than the SIMPLE schedule, low system load case (see Figure 17). Three of the processes have finished their loop iterations, this is indicated by vertical lines on the left edge of the graph that reach the count value of 100. The remaining process did not complete its work until much later. The Time increases to the right and the number of counts is indicated on the vertical axis.

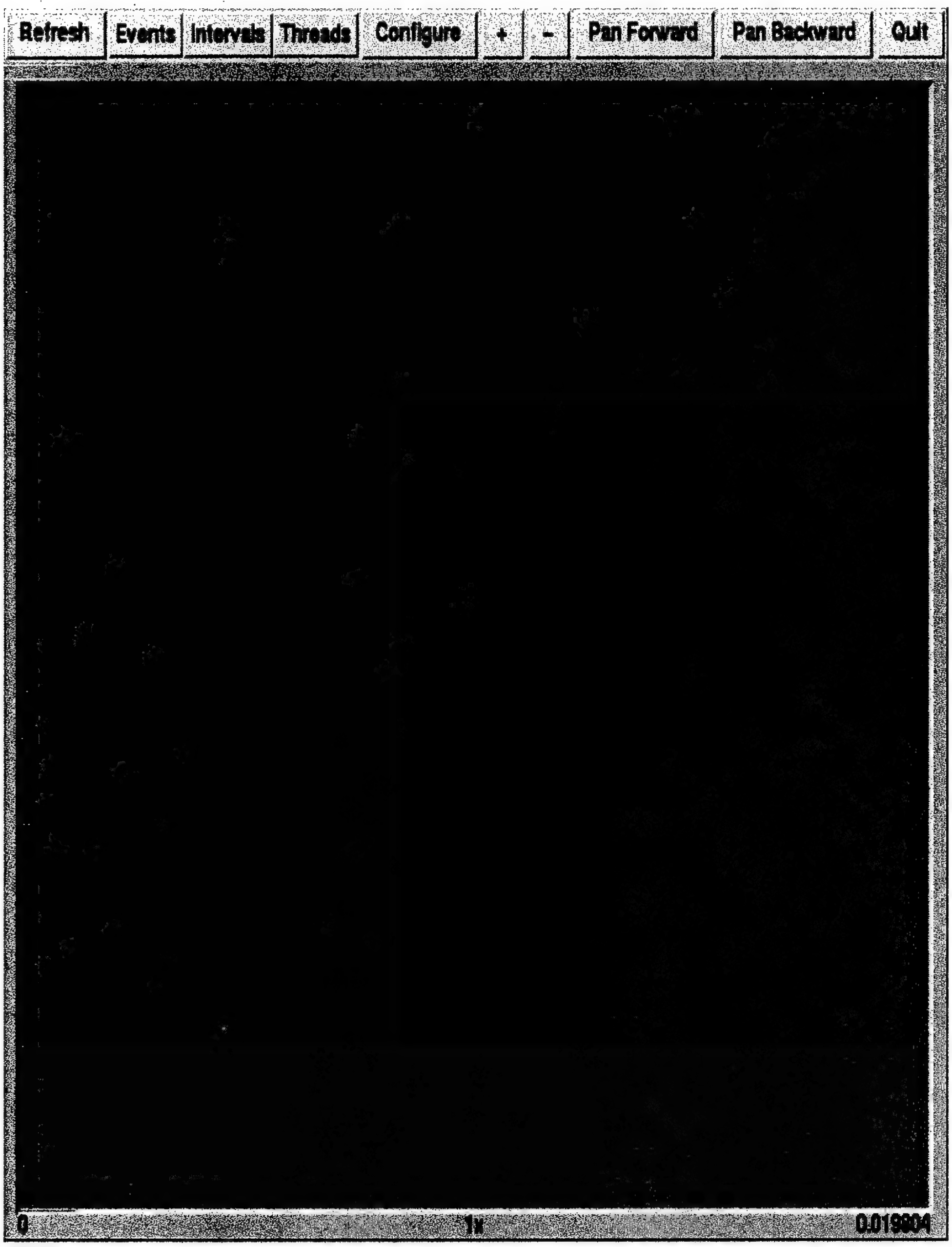

Figure 28. Plots of event_y for 4 lightweight processes using SIMPLE scheduling under a high system load condition (from the same data as in Figure 27). Each event_y denotes the completion of a single loop iteration. The Time increases to the right. The clusters of events on the left edge of the graph show that Processes 0, 1, and 3 have finished their portion of loop iterations, where as the process 2 did not complete its portion of loop until much later, as shown by the cluster of events on right edge of the graph.

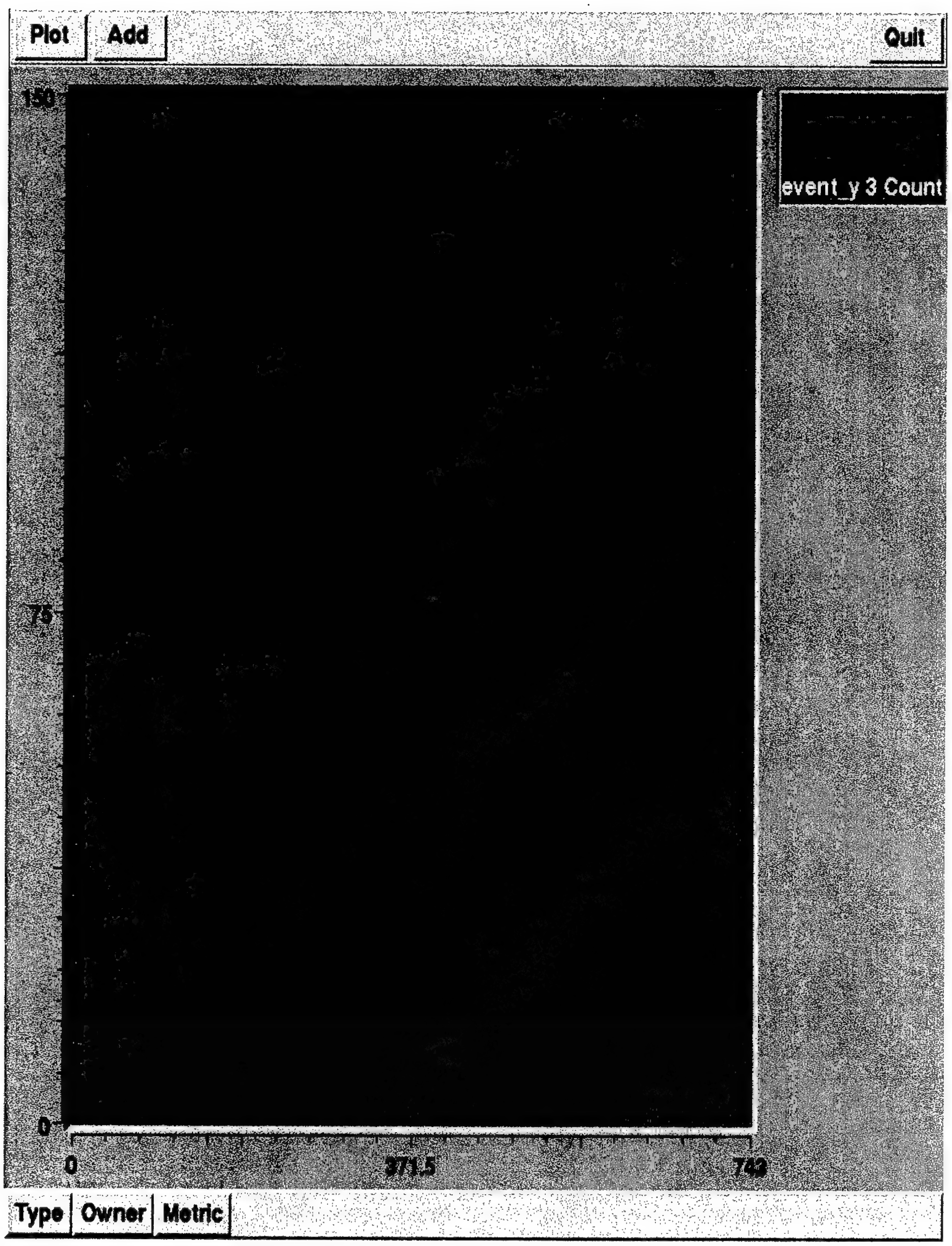

Figure 29. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using DYNAMIC scheduling under a high system load condition. The Time increases to the right and the number of counts is indicated on the vertical axis. Although the workload is not evenly distributed among the processes, even under the high load condition, they finish about the same time.

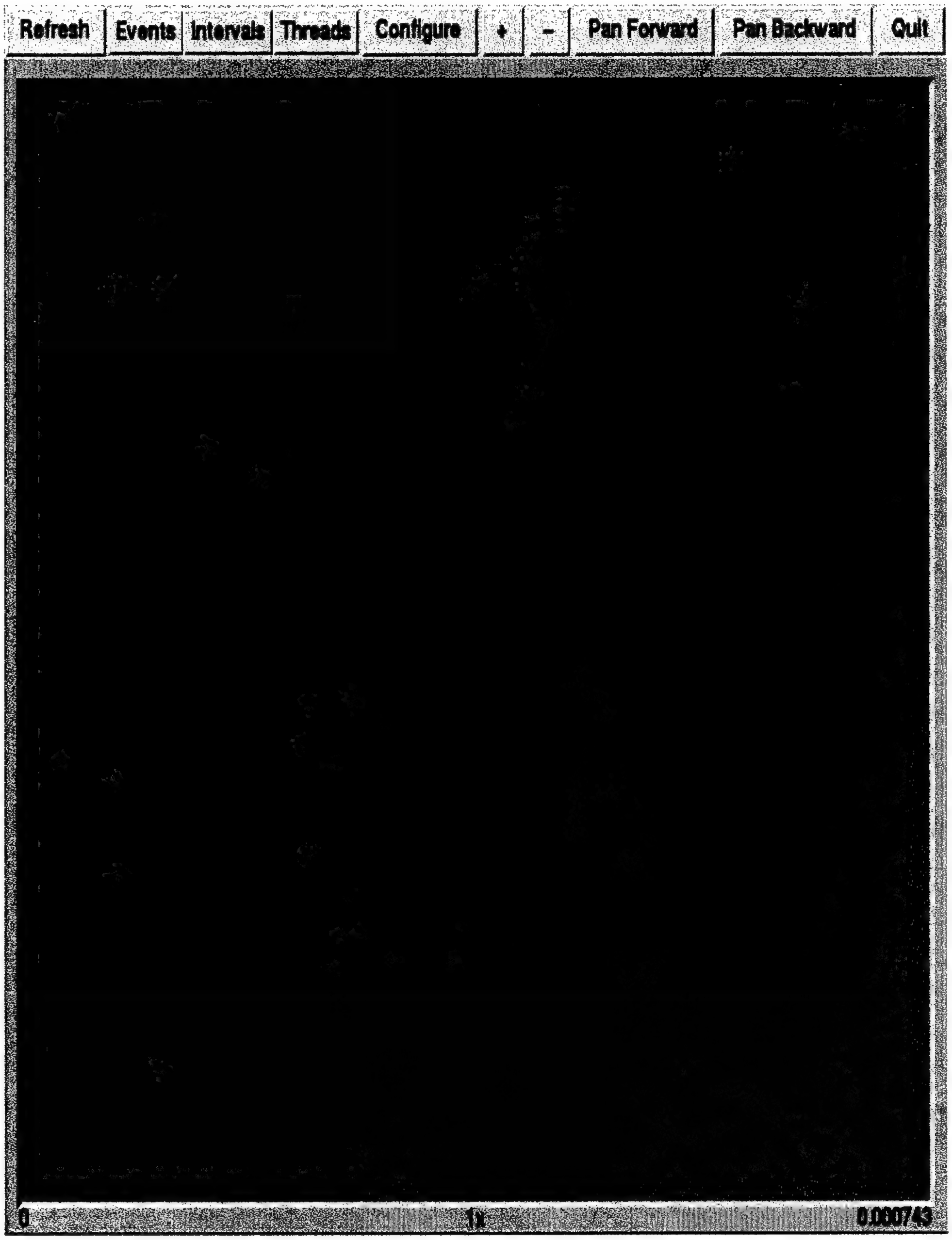

Figure 30. Plots of event_y for 4 lightweight processes using DYNAMIC scheduling under a high system load condition (from the same data as in Figure 29). Each event_y denotes the completion of a single loop iteration. Although we request 4 processes, only 3 processes participated in the parallel region due to high system load.

program running with the GSS scheduling policy under high load condition does not degrade significantly. This is not the case with the example program running with the INTERLEAVE schedule type under the high load condition (Figure 33 and 34). Like the SIMPLE schedule type under the high load condition, the example program running with the INTERLEAVE scheduling policy can experience a significant slow down if one of the participating processes is blocked. The remaining processes must wait until the last process finishes. As shown in Figure 34, process 0, 1, and 2 spend a large portion of time idling.

The results from running the example FORTRAN program with different workload schedules indicates that the DYNAMIC and GSS scheduling type have better overall performance characteristics as the system load increases, at least when the computation is simple. The strategy of allowing processes to grab more chunks of workload (i.e., be more greedy) as they become available for work is a better overall strategy. It minimizes the amount of wait time when the system load is high and other processes are competing for the available CPU resources.

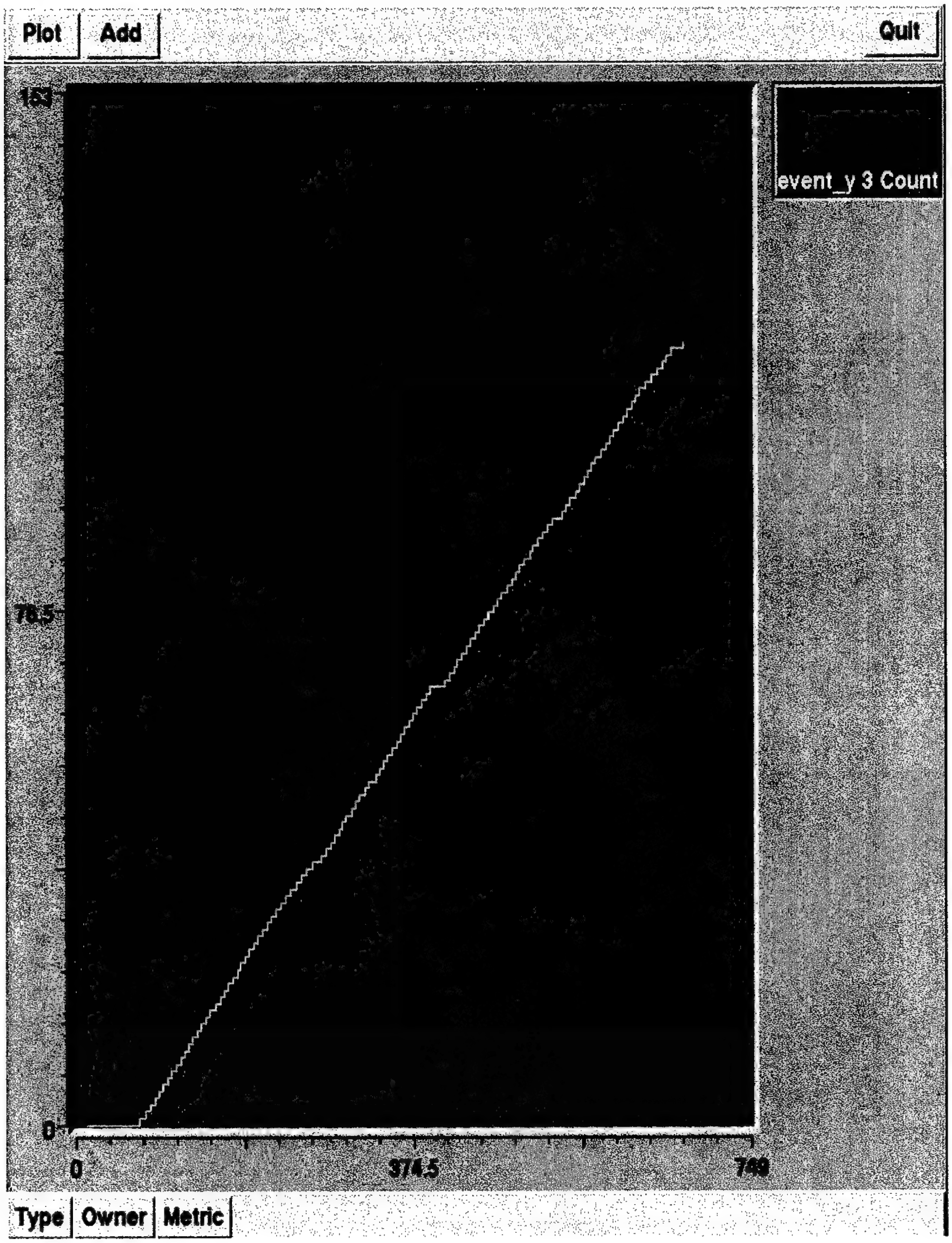

Figure 31. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using the GSS schedule type under a high system load condition. The Time increases to the right and the number of counts is indicated on the vertical axis. Although we request 4 processes, only 3 processes actually participated in the parallel region. The workload is not evenly distributed among the processes but they finish about the same time.

Figure 32. Plots of event_y for 4 lightweight processes using GSS scheduling under a high system load condition (from the same data as in Figure 31). Each event_y denotes the completion of a single loop iteration. Although we request 4 processes, only 3 processes (0, 1, and 3) actually participated in the parallel region due to high system load.

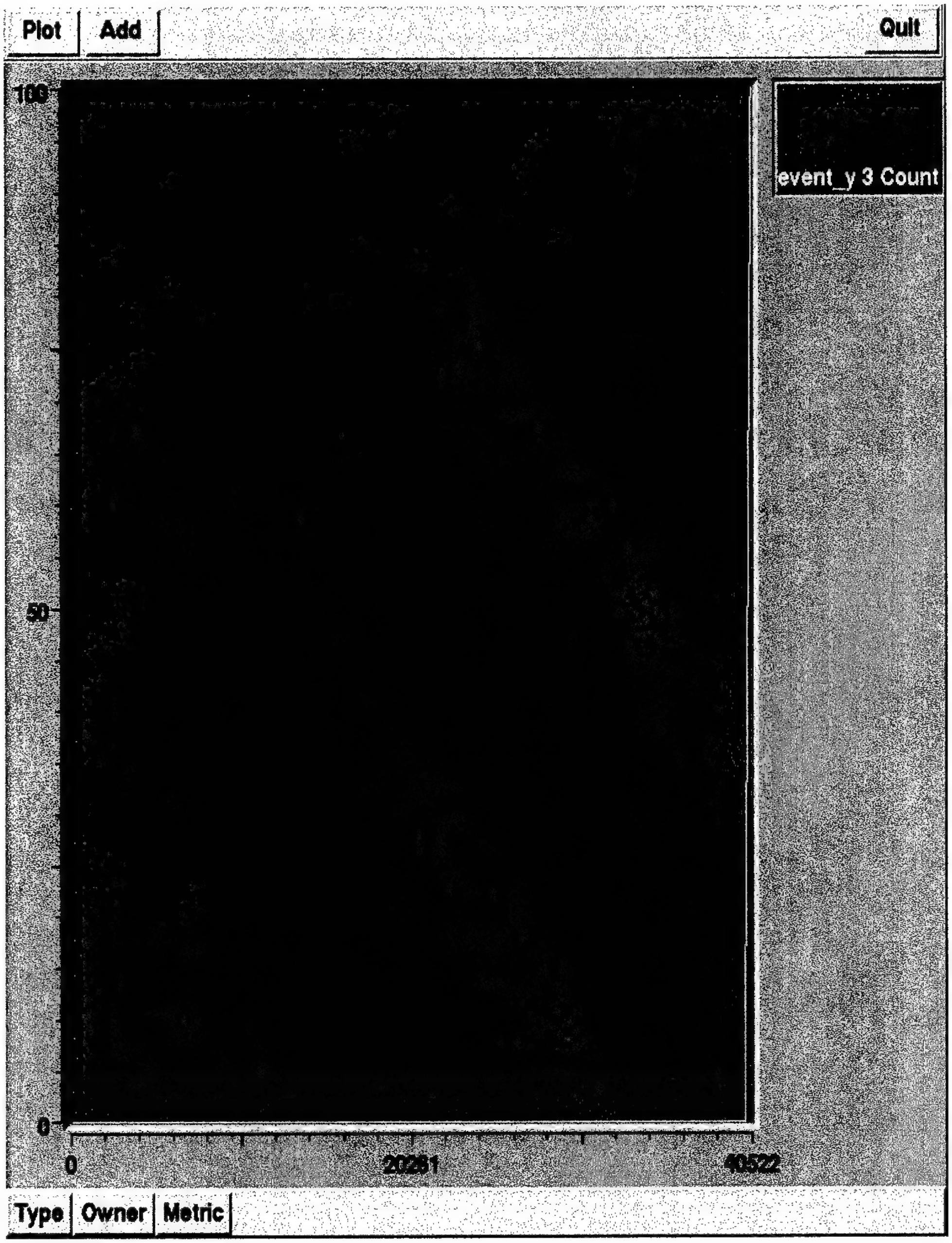

Figure 33. Plots of the counts of event_y as a function of time. The plots are for 4 lightweight processes using the INTERLEAVE schedule type under a high system load condition. Each event_y denotes the completion of a single loop iteration. The execution pattern of this case is similar to SIMPLE schedule under a high system load condition. (see Figure 27). Three of the processes have finished their loop iterations, this is indicated by vertical lines on the left edge of the graph that reach the count value of 100. The remaining process did not complete it work until much later. The Time increases to the right and the number of counts is indicated on the vertical axis.

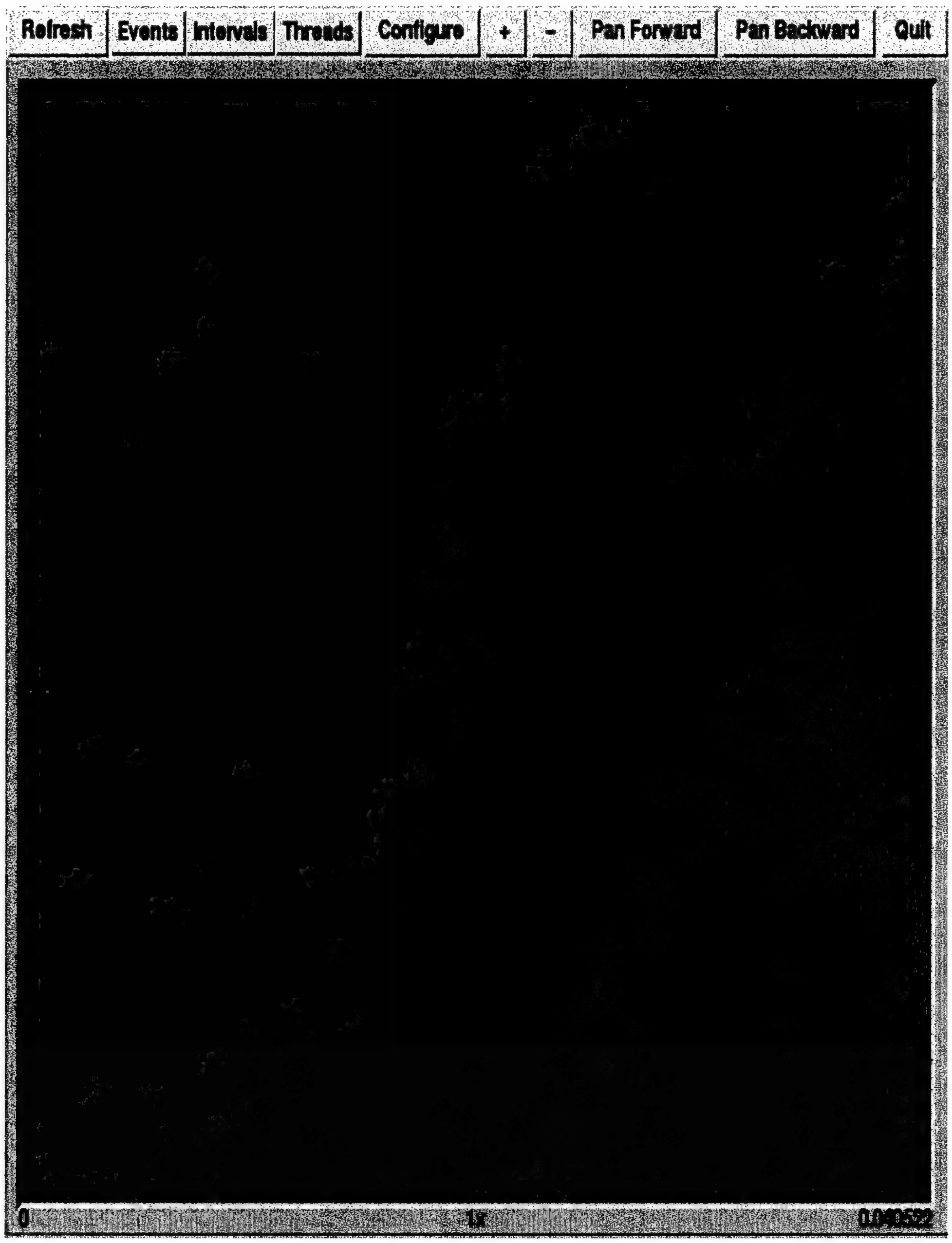

Figure 34. Plots of event_y for 4 lightweight processes using INTERLEAVE scheduling under a high system load condition (from the same data as in Figure 33). Each event_y denotes the completion of a single loop iteration. The Time increases to the right. The clusters of events on the left edge of the graph show that Processes 0, 1, and 2 have finished their portion of loop iterations, where as the process 3 did not complete its portion of loop until much later, as shown by the cluster of events on right edge of the graph.

4. Summary

In this chapter, we discussed the changes we had to make to *Glimpse* for it to monitor automatically parallized FORTRAN programs. In particular, we show that the language independent concepts (i.e., event and interval) and the Data Collection Facility of *Glimpse* is easily adapted to monitor automatically parallelized FORTRAN program.

Using the modified *Glimpse*, we evaluate the performance of a parallel example FORTRAN program with different workload scheduling algorithms under different load conditions. The results indicate that dynamic scheduling schemes (i.e., DYNAMIC and GSS schedule), although having higher overhead due to workload distribution, can perform better when the system load is high. Other scheduling policies performance can degrade significantly under the high system load condition when one or more participating processes is blocked due to competition with other processes.

VII. SUMMARY

In this thesis we describe the design, implementation, and testing of a language independent performance debugging tool called *Glimpse*. The *Glimpse* toolkit collects event data from the execution of multi-threaded programs and provides visualization tools to help programmers analyze the collected data. This work builds upon the approach taken by *Graze*, a performance debugging tool that monitors C++ programs that use the Solaris thread library, and extends it to applications using Java threads and to automatically parallelized FORTRAN programs.

1. Future Work

Before summarizing our experiences with *Glimpse*, we describe some future work areas that will enhance *Glimpse*.

a. *Adding generalized math expressions to the interval definition*

In chapter IV, we describe the generalization of the *interval* definition to allow for partial matching of attribute values and Boolean relationships between the event's attributes. An example of this interval definition is:

```
interval Transit[ s:Sendmsg -> r:Recvmsg ]
         { s.src=r.src && s.msgsize = r.msgsize } = line;
```

In this example, interval `Transit` requires the attribute value `src` of Sendmag and Recvmsg to match, and that the `msgsize` value of the `Sendmsg` event be equal to the `msgsize` value of the `Recvmsg` event. It would be useful to add generalized mathematical expressions to the interval syntax to further generalize the interval definition. This capability would allow the user to define intervals by specifying certain threshold values. For example, to define intervals where the length of message sent is greater than 100 words, we would write

```
interval Transit[ s:Sendmsg -> r:Recvmsg ]
         { s.src=r.src && s.msgsize/WORDSIZE > 100 } = line;
```

b. Enhancement to the visualization tools

Currently, the interval definition is defined in the specification file. This file is read by the visualization tools, *gorge* and *nibble*, during start up. To visualize the event data with a different interval definition, the user needs to modify the specification file and restart the visualization tools in order for it to display the changes to the interval definition. It would be useful to add the capability, in the visualization tools, to allow the user to create new interval definitions, and to edit the existing interval definitions on the fly. This would make it less cumbersome for the user to experiment with different interval specifications.

Another enhancement to the visualization tools is to add the capability to associate the log data being displayed with the thread name from which it came. Recall from the discussion in chapter IV, each thread is mapped to a unique log file by the Data Collection Facility. In the current implementation, the logical thread name used by the program during execution is not saved to the log file. Therefore, to associate data from a log file back to a particular thread requires some careful interpretation of plots and understanding of the program code by the user. To make this process easier for users, we should modify the Data Collection Facility to save the logical name of the thread along with the data to the log file. The visualization programs can then use this information to label the plots with the thread names.

c. Improving the performance of Glimpse's visualization tools for large data sets

After we successfully tested the FORTRAN version of *Glimpse* with various workload scheduling policies (chapter VI), we applied it to a Navy Operational Weather Forecast application. This application, COAMPS, is a weather simulation program written in FORTRAN. The current operational version of COAMPS relies heavily on the compiler-directive approach to optimize the runtime performance. However, we noticed that its performance does not scale well in the multiprocessor environment. For example, the maximum speedup of this application when running

on SGI multiprocessor systems is approximately 2.6 times that of the single processor mode. This maximum speedup value is observed when the COAMPS application is running with 4 processors.

We instrumented this application with *Glimpse* in an attempt to identify the possible performance bottlenecks. The instrumented code did not add any significant overhead to the COAMPS run time; the instrumented code is perhaps 2 to 3 percent slower than the non- instrumented code. The data volume generated by the instrumented code was about 50 mbytes (approximately 250,000 events) for a typical COAMPS run. The *gorge* tool took almost an hour to load the event data and display the intervals. The slowness appears to be related to the algorithm that is used to build the interval instances from the event data stream. Given a number of event instances in the event stream n, this algorithm finds all instances of an interval type by starting the search at the head of the event stream and moving forward until it finds a matching ending event. This algorithm performs this operation n times, each time starting the search one position forward in the event stream. Thus, for the worst case scenario, the total number of operations needed to find all instances of a single event type is given by $(n\text{-}1) + (n\text{-}2) + (n\text{-}3) + \ldots + 2 + 1$. This summation reduces to $n * (n - 1) / 2$, which is on the order of n^2 for large n. Further investigation is needed to improve the performance of this algorithm for handling large event data sets.

2. Experiences using *Glimpse* with concurrent Java and FORTRAN programs

We found that the `event` and `interval` concepts, together with visualization tools that can display them, are very useful for programmers. It helps them to understand the behavior of parallel applications, independent of the language environment. The glimpse's *event* and *interval* concepts are similar to that of the *event* and *state* used in the commercial software, *TimeScan* (see chapter II). However, in TimeScan the event definitions are defined in the source code, whereas *Glimpse* uses a speci-

fication file to store the user-defined events. The specification file approach is more flexible and minimizes the changes to the source code of the program that the user wants to monitor. Another important difference between *TimeScan* and *Glimpse* is that *TimeScan* does not support an interval that spans two threads. *Glimpse* does not have this limitation.

Naturally, the process of porting *Glimpse* to the Java and FORTRAN environment, and to other languages or programming environments, requires writing a platform-specific Data Collection Facility that contains such utilities as functions to handle input/output, timestamps, a utility program to translate event specifications to data logging functions, and to handle other low level issues. From the perspective of the user of the *Glimpse* tool, even such implementation differences are hidden. The user only needs to concentrate on understanding the behavior of the program, and describing those behaviors using events and intervals.

APPENDIX A. OVERVIEW OF JAVA AND JAVA THREAD

This appendix presents a brief overview of Java and the mechanisms that it provides for building multi-threaded programs.

a. Overview of Java

Java is an object-oriented programming language developed by Sun Microsystems [Ref. 20]. One of its design goals is to allow programs to run on different type of operating systems and architectures without recompilation or any other modification. To achieve this goal of platform independence, the Java specification describes not only the programming language itself, but also the Java Virtual Machine (JVM) [Ref. 21] and Application Programming Interface (API). The concept of the Java Virtual Machine is similar to that of a virtual machine operating system. It defines the interface and functionality of the layer between Java programs and that of the underlying architecture. A Java program is compiled into bytecodes, which are instructions understood by the JVM, and stored in class files. The class files are then loaded and executed by the Java Virtual Machine. To execute a Java program, a Java Virtual Machine needs to provide support for (i) memory allocation, which includes garbage collection, (ii) dynamic loading and verification of class files, (iii) index checking on array access, and (iv) threads. It also provides an interface to native functions such as those that support network and windowing environment operations. Currently most implementations of the Java Virtual Machine are implemented in software and thus depend on resources provided by the underlying native operating system. Consequently there is some overhead and performance penalties associated with executing bytecodes compared with executing typical native machine code.

In addition to the Java Virtual Machine, the Java specification defines a comprehensive API specification. In the Java version 1.1, there are approximately 500 classes for the core library alone, and this number is expected to get larger in the

next release. The core Java API, in essence, defines the software infrastructure that should be available on all platforms that support Java. The current API includes packages for graphical user interface components, thread libraries, input and output streams, data structures, database access and internationalization of applications.

By providing a high-level and uniform API across various platforms, the Java environment isolates programmers from the variations and differences in functionality and interfaces provided by different operating systems. For example, Java provides a Thread class (java.lang.Thread) that contains constructors for creating a thread and methods for setting the priority of a thread, suspending a thread, naming a thread and getting information about a thread's status. This package has the same interface on both POSIX and Window NT platforms. This standardization of interface permits programmers to focus on building programs rather than on learning another programming interface. It is the job of the commercial vendors and the freeware and open-source community to provide an efficient and correct implementation of the Java API library and JVM for their respective platforms.

In section (b.), we will describe the synchronization mechanism that is built into the Java programming language, specifically, the monitor object, and how it is used to ensure that a class method can have exclusive access to a variable or another object in a Java program. Additionally, the wait and notify mechanism provided by Java to transfer execution between threads is discussed. In the conclusion section, we will summarize a few simple rules for using synchronization in a Java program.

b. Java Threads and Synchronization

The Java language is designed to support concurrent programming, with the thread as the basic unit of concurrency. A concurrent Java program can have many threads of execution. In some implementations of the Java Virtual Machine, threads can execute on a separate processor if the underlying hardware is a multiple-processor system. In such systems the execution of threads can overlap in time. On single-processor systems, the Java Virtual Machine supports concurrent execution by inter-

leaving different threads to give the appearance of simultaneous execution.

A fundamental requirement for support of concurrent programming in any language is the ability to provide synchronized access to a shared variable or to any of a set of related critical sections. In many concurrent programs mutual exclusion is the technique used to ensure proper synchronization among threads. The following discussion will focus on the language features provided by the Java programming language for implementing synchronization using mutual exclusion.

A typical multi-threaded Java program consists of cooperating threads working on some tasks, where each thread has its own program counter and stack. Within the context of each execution thread, program instructions are carried out in sequential order. Variables that are used, or assigned to, while each thread executes can be considered as either residing in local or shared memory. Each thread has its own copy of local variables, which are not visible to any other threads. A shared variable, on the other hand, is visible to any thread that has a reference (i.e., a memory address) to that variable. Cooperating threads can only pass information to one another through shared variables. Therefore, the issue of synchronization in general, and specifically mutual exclusion, is only of concern for accessing variable in shared memory locations.

The simple Java program below can be used to illustrate that simultaneous access to shared variables, without proper synchronization, could give an incorrect result. The program models the transaction between customers and their bank accounts.

```
class bank_account {

    /* The current dollar amount in the customer account */
    int balance;

    /* constructor, initialize account. */
    bank_account(int initial_deposit) {
        balance=initial_deposit;
    }
```

```
    void Deposit(int deposit) {
        balance=balance+deposit;
    }

    void Withdraw(int amount_to_withdraw) {
       /* check for sufficent funds in account */
       if (balance >= amount_to_withdraw)
       {
          balance = balance-amount_to_withdraw;
       }
       else
       {
          /* issue warning about account overdrawn */
          System.out.println("Account overdraft");
          return;
       }
    }
}
```

The **bank_account** class has two methods: the **Withdraw** method that allows the customer to remove money from the account and the **Deposit** method that lets the customer put money in. The account is initially credited with some dollar amount when it is first created. To ensure overdraft does not occur, the **Withdraw** method first verifies that the account has sufficient funds to cover the transaction. If there are insufficient funds, a warning message is issued and the **Withdraw** method returns without changing the balance. Otherwise the balance of the account is reduced by the amount withdrawn.

To simulate customer activity on the bank account, we let each customer's transaction be represented by a thread. Suppose two customers try to withdraw money from the same account. The **Withdraw** method of the **bank_account** object is invoked twice, once by each separate threads. Each thread uses a different copy of the variable **amount_to_withdraw**, but modifies the same copy of the variable **balance**, i.e., the variable **amount_to_withdraw** resides in the local memory of each thread whereas the variable **balance** resides in the shared memory. In summary, more

than one thread can be executing in the same instruction code segment, but they each have their own program counters and local working memory, and can access or modify variables shared with other threads.

Before describing why synchronization is important, we need to look at how a JVM schedules threads for execution. In the context of scheduling, a Java thread that is not currently executing is either in a *runnable* or *blocked* state. Typically, the *runnable* state is the default state for a thread unless it has performed an operation that causes it to become blocked. For example, a thread that performs a network-related operation such as reading from a socket when the data for that socket is not yet available will become blocked. When the data becomes available the thread will become runnable again. In a multi-threaded Java program, every thread is competing for CPU cycles and it is up to the JVM to select a thread from among the pool of *runnable* threads to be the currently running thread. JVM uses a priority-based preemptive scheduling mechanism that guarantees the currently running thread always has a priority value equal to the highest priority value found among all *runnable* threads.

A thread will remain as the currently running thread until a scheduling event signals the JVM to select another thread from the pool of *runnable* threads as the current running thread. There are three types of scheduling events: (1) when a currently running thread becomes blocked or completes; (2) when a higher priority thread enters the *runnable* state; and (3) when a timer interrupts execution. In the case of scheduling event (1) or (2), the currently running thread yields the CPU and the JVM selects another runnable thread. Not all implementations of a JVM must support the scheduling event (3). On most UNIX platforms only the first two scheduling events are supported, whereas on the Windows NT operating system, the timer-based scheduling event is also supported. A multi-threaded Java program will behave differently depending on the JVM scheduling events. For example, consider a Java program that has two long running threads of equal priority, and assume that

neither thread performs an instruction that will cause it to become blocked. On UNIX systems, if thread 1 is started first and never enters the blocked state, it will continue to run until it is completed. When thread 1 is completed, it leaves the *runnable* state, thus creating a scheduling event, at which time thread 2 becomes the currently running thread. Running the same program on a JVM that supports timer-based scheduling, thread 1 starts first and at some time later a timer-based scheduling event occurs. Thread 2 becomes the newly running thread and is executed by the CPU until the next timer-based scheduling event, at which time thread 1 becomes the running thread again. Under the timer-base scheduling scheme, thread 1 and 2 alternate as the currently running thread (i.e., similar to a round-robin scheduler), with each thread getting a time-slice of CPU cycles. Thread 2 might even finish before thread 1.

From the perspective of a running thread, it does not know when a scheduling event will occur. The switching of execution context from one thread to another thread is performed by the JVM; a running thread can be interrupted at any time as the result of any of the scheduling events described above. We now return to the synchronization issue and our example program. We will assume the **bank_account** program is running on a JVM with timer-based scheduling.

In the following scenario, suppose the bank account is a joint account and that two people attempt to obtain cash from their joint account at the same time. For example, the husband (thread A) is withdrawing $80 and the wife (thread B) is withdrawing $50 from the same account. Assume the initial balance is $100. If after thread A executed the first line of code

```
(if (balance >= amount_to_withdarw)),
```

but before it can execute the next line of code to subtract the amount from the variable **balance**, a timer-based scheduling event occurs. The JVM will now select the next thread from the *runnable* thread pool with the highest priority value as the current running thread. Since both thread A and B have the same priority, the JVM

will move thread A to the *runnable* pool and make thread B the currently running thread. Thread B executes the following line of code

```
(if (balance >= amount_to_withdarw)).
```

The variable **balance** is still at \$100 because thread A was interrupted before it could modify the value of **balance**. Thread B continues to execute and subtracts \$50 from the joint account, and the variable **balance** is updated to reflect the new value of \$50. Thread B finishes execution by exiting the **Withdraw** method and a scheduling event is generated. The JVM selects thread A as the new running thread. Thread A resumes execution at the following line of code:

```
balance=balance-amount_to_withdraw;
```

The variable **balance** has the value of \$50 at this point, and now thread A subtracts \$80 from \$50. The new value of the variable **balance** is negative \$30 dollars, which is clearly incorrect. This error results from lack of synchronization when multiple threads try to access and modify a shared variable. In this example, a mechanism is needed to provide proper synchronization to the shared variable **balance**. Such a mechanism would immediately block thread B when it tries to execute the **Withdraw** method, because thread A was already in this same method. This type of synchronization is called mutual exclusion. Mutual exclusion ensures that only one thread can enter a particular code segment or method that modifies and uses the same shared variable. These code segments or methods are called critical sections. If mutual exclusion is used in this example, when thread B tries to execute the **Withdraw** method, it will be blocked and the JVM will re-select thread A for execution. After thread A completes, thread B is unblocked and is made the currently executing thread by the JVM. Thread B now sees variable **balance** has a value of \$20 and realizes that there are insufficient funds for withdraw.

Java supports mutual exclusion with a mechanism known as a monitor. The concept of the monitor was first introduced by Brinch Hansen [Ref. 22, 23]. A

monitor acts as a lock to a section of program code, allowing only a single thread to enter that code segment at a time. Any object in Java can be a monitor. To place a block of code under the auspices of a single monitor, the Java language provides a reserved keyword *synchronized.* For example, to ensure that variable **balance** is properly updated when accessed by different threads, we modify the **bank_class** by adding the *synchronized* statement to the block of code that encloses the variable **balance.**

```
void Withdraw(int amount_to_withdraw)
{
   synchronized(this) { /* Acquire monitor, beginning of the
                           critical section, use 'this' object
                           as the monitor */

       if (balance >= amount_to_withdraw) /* check for sufficent
                                             funds in account */
       {
           balance = balance - amount_to_withdraw;
       }
       else
       {
           /* issue warning about account overdrawn.*/
           System.out.println("Account overdraft");
           return;
       }
   } /* Release monitor, end of the critical section */
}
```

The statement *'synchronized (this)'* marks the beginning of the critical section to be placed under the monitor. The token following the 'synchronized' is the reference to the object acting as the monitor for this code segment. In this example *'this'* refers to the object which the **Withdraw** method belonged to. Other objects can also be used as the monitor in place of *'this.'* For instance, we could have defined a string object in the class declaration of **bank_account** and used that string object as the monitor and it would have the same effect.

```
class bank_account {

    /* Monitor object for each instance of the bank_account,
       each account has its own monitor. */
    String customer;

    /* The current dollar amount in the customer account */
    int balance;

    /* constructor, initializes the customer account. */
    void bank_account(String customer_name, int initial_deposit)
    {
        customer=new String(customer_name);
        balance=initial_deposit;
    }
    .
    .
    void Withdraw(int amount_to_withdraw)
    {
        synchronized(customer) /* Acquire monitor at the
                                  beginning of the critical
                                  section, use the string
                                  variable 'customer'
                                  as the monitor. */
        {
           if (balance >= amount_to_withdraw) /* check for sufficient
                                                 funds in account */
           {
               balance = balance - amount_to_withdraw;
           }
           else
           {
               /* issue warning about account overdrawn */
               System.out.println("Account overdraft");
               return;
           }
        } /* Release monitor, end of the critical section */
   }
}
```

Before the current thread (the thread that the CPU is currently executing

instructions for) can execute code inside the critical section, it must first enter the monitor guarding that critical section. If another thread has already entered any critical section protected by the same monitor, then the current thread must wait until the other thread exits the monitor. Many critical sections can be associated with the same monitor, but only one thread can be granted that monitor at any one time. In other words, if one thread is executing in a critical section guarded by a monitor, than no other threads can enter other critical sections guarded by the same monitor. Conversely a single thread can enter more than one monitor if need be. For instance, a thread can execute in one critical section guarded by monitor A and then enter another critical section guarded by monitor B, provided that no other threads have entered monitor B. This is perfectly legal as the mutual exclusion mechanism in Java is based on scope of the monitor object.

If a thread has been granted the ownership of a monitor, and tries to acquire the same monitor again the thread would not be blocked. This situation arises when the currently executing thread, while inside a critical code section tries to execute another code segment guarded by the same monitor. For example, a thread making recursive calls from a synchronized method. The design of Java monitors allows the same thread to re-enter the critical section once it acquires the monitor, or to enter other critical sections guarded by the same monitor. This re-entrant behavior of the monitor eliminates the possibility of a thread waiting on itself.

Returning to the bank account example, we consider the situation that occurs when one customer tries to withdraw more money than is available in the account. Instead of returning with an error message, we may prefer that the customer wait until another customer has deposited additional money into that account to cover the withdrawal. This is a very common pattern in many multi-threaded programs where a thread executing in a critical section might encounter a condition such that it cannot proceed until another thread can act to change that condition. What is needed to support this type of programming construct is a mechanism to transfer

the execution from one thread to another thread in the critical section. In order for this to work, the currently executing thread must temporarily halt and give up its claim to the monitor so that another thread can execute in that critical section. Once the condition is satisfied by the action of another thread, the original thread can then proceed again. The Java language provides a wait and notify mechanism for this purpose. Of course, the inclusion of wait and notify are necessary to make the mechanism that we have been discussing truly qualify as a monitor. Without them, semaphores may have been included in the language and would likely have been more efficient.

To see how wait and notify work, we modify our bank account program so that when a customer overdraws an account, he does not receive an error message but instead merely waits until another customer deposits more money into that account.

```
class bank_account {

    /* Monitor object for each instance of the bank_account,
       each account has its own monitor. */
    String customer;

    /* The current dollar amount in the customer account */
    int balance;
    .
    .
    void Deposit(int deposit)
    {
        synchronized (customer)
        {
           balance=balance+deposit;
           customer.notifyAll();    /* Notify any thread waiting on
                                       this monitor */
        }
    }

    void Withdraw(int amount_to_withdraw)
    {
       synchronized (customer)        /* Acquire monitor */
       {
```

```
            while ( true )
            {
               if (balance >= amount_to_withdraw) /* check for
                                                     sufficient fund */
               {
                  balance = balance - amount_to_withdraw;
                  return;
        }
               else
               {  /* Relinquish monitor and put the current
                     thread on the wait queue of this monitor.
                     Catch interrupted exception if this thread
                     unexpectedly returns from wait() method. */
                  try { customer.wait(); }
                  catch (InterruptedException e) { System.exit(1); }
               }
            }
        } /* End of critical section */
     }
  }
```

The modified code works as follows. If a thread tries to withdraw more money than what is in the specified bank account, it will be suspended from execution (i.e., it is blocked in the **wait()** method call) until some other thread has called the **Deposit()** method. After another thread deposits money into the account, it calls the **notifyAll()** method before exiting the **Deposit()** method. The **notifyAll()** method sends a signal to all threads that are blocked in the **wait()** method. These threads are removed from a wait queue and placed in the ready queue by the JVM. When a blocked thread returns from the **wait()** method it checks whether there are sufficient funds in the account. If not, it calls the **wait()** method again and repeats the process until other threads have deposited sufficient money to cover the withdrawal. The **wait()** and **notifyAll()** can only be called from inside a monitor, i.e., only the thread currently inside the monitor can invoke the **wait()** and **notifyAll()** methods of that monitor.

Unlike the **wait()** method, the **sleep()** or the **yield()** method (from the

java.lang.Thread library) has no effect on the state of the monitor when invoked. It merely tells the JVM to select another thread as the currently running thread. When the **wait()** method is invoked, it places the current thread on the wait queue of the monitor and relinquishes mutual exclusion prior to suspending the current thread. The **notifyAll()** method releases all threads waiting on a condition in that same monitor. The **wait()** and **notifyAll()** methods are not part of **java.lang.Thread** library but are an integral part of the Object class and are tightly integrated into the JVM. They provide a reliable mechanism to safely transfer the control from one thread to another thread in a monitor.

Suppose instead of calling the wait() method, a thread invokes the **sleep()** or **yield()** method while inside the **Withdraw()** method to temporarily suspend itself so that another thread can be executed by the JVM. In this example, calling the **sleep()** or **yield()** method would lead to a dead-lock situation when executing on a machine with a non-preemptive thread scheduler: one customer starts to perform a withdraw action but cannot complete that transaction until more money is deposited by another customer. However the deposit action by another customer cannot begin until the **Withdraw** method is completed because the depositing thread would not be permitted to enter the monitor.

The Java language provides the basic synchronization primitives via the monitor and the wait and notify mechanisms. Other, higher-level, synchronization facilities can be built on top of these basic primitives. For example, counting semaphores, can be built from the **wait()** and **notifyAll()**. In the basic wait and notify mechanism if a **notifyAll()** method is called when there are no threads waiting on a monitor, then that notification event has no effect. Counting semaphores, however, maintain state information that indicates how many times **wait()** or **notifyAll()** are called. When one thread calls the **notifyAll()** method and there are no threads waiting, the notification event causes a value to be incremented in the data structure of the associated semaphores. Counting semaphores are useful for providing synchroniza-

tion to a shared buffer in a producer-consumer scenario. Similarly, other higher-level synchronization facilities such as those needed to solve the multiple reader, single writer problem can be constructed using the basic primitives provided by Java.

c. Summary of Java Threads and Synchronization

In this appendix, we reviewed the support for writing multi-threaded program provided by Java and some basic concepts for synchronization. These include:

a. Cooperating threads that communicate via shared variables. If the correctness of program execution depends on the order in which threads act on the shared variable, then access to these variables needs to be synchronized.

b. Java provides an elegant and simple mechanism for synchronization. To ensure synchronized access to shared variables, code segments that act on shared variables should be placed inside a synchronized block or method.

c. Before a thread can execute a synchronized block or method, it first acquires the monitor associated with that block or method. A monitor can be associated with multiple synchronized blocks or methods. However, only one thread can execute inside a synchronized block or method of a given monitor at any time.

d. To transfer control of program execution between threads from inside a synchronized block or method, a program should use **wait()** and **notifyAll()** methods provided in the monitor class. These methods guarantee that the monitor is relinquished when a thread is placed in a wait state, and that the monitor is re-entered prior to resuming execution.

e. Other forms of synchronization techniques such as the counting semaphore for controlling multiple access to limited resources, and mechanisms that can be used to solve such commonly occurring problems as the multiple-reader, single-writer can also be constructed using the basic monitor mechanism provided by Java.

APPENDIX B. GRAMMAR OF THE GLIMPSE SPECIFICATION LANGUAGE

This appendix describes the grammar for the *Glimpse* specification language. The italicized words are non-terminals; words enclosed by single quotes are literals. Tokens that denote keywords used by the language are prefixed with `k_`. The list of keywords is given in section two.

a. Grammar of the Glimpse Specification Language

The goal of specification language:

spec ::= k_glimpse *ident events intervals* k_end '.'

`Event` has a name, associated attributes, and an optional graphic symbol.

events ::= NULL | *event events*

event ::= k_event *ident memberlist* '=' *symbol* ';' | k_event *ident memberlist* ';'

`Event` can have 0 or more attributes; attributes are separated by commas and are enclosed by a left and right parentheses.

memberlist ::= NULL | '(' *members* ')'

members ::= NULL | *ident* | *ident* ',' *members*

intervals ::= NULL | *interval intervals*

An `interval` has a name, a start event, search direction, an ending event, a matching criteria, and a graphic symbol.

interval ::= k_interval *ident* '[' *ident search_dir ident* ']' *match_type* '=' *connection_type* ';'

A second form of `interval` production is presented below. This production allows a boolean expression for matching the start and end events.

interval ::= k_interval *ident* '[' *ident* ':' *ident search_dir ident* ':' *ident* ']' *where_clause* '=' *connection_type* ';'

search_dir ::= ',' | k_forwards | k_backwards

The production below specifies the matching criteria for the start and end event. If the match_type is NULL, then matching is done based on the start and end event type matching only these events that occur within the same thread. Otherwise, the matching is done on the attribute values.

match_type ::= NULL | k_match

The logical expression for matching the start and end events is shown below.

where_clause ::= NULL | " *expr* "

expr ::= *relation* | *relation* k_and *expr*

relation ::= *ident* '.' *ident math_oper ident* '.' *ident*

math_oper ::= k_lt | k_le | k_eq | k_ge | k_gt | k_ne

connection ::= k_connect '(' *connection_type* ')'

Graphics symbols for `event` and `interval` types are shown below.

symbol ::= *symbol_type offset* | k_symbol '(' *symbol_type* offset ')'

symbol_type ::= NULL | k_x | k_plus | k_box | k_diamond

connection_type ::= NULL | k_line | k_rectangle

offset ::= NULL | ':' ident

ident ::= k_ident

b. Keywords for the Glimpse Specification Language

The keywords used in the *Glimpse* specification language are on the right hand side of the statements below, shown as regular expression. As in the specifications for `lex`, [a-zA-Z] means any character between `a` and `z` or between `A` and `Z`. The * represents the Kleene closure.

k_graze ::= `graze` | `Graze`

k_event ::= `event` | `Event`

k_end ::= `end` | `End`

k_symbol ::= `symbol` | `Symbol`

k_connect ::= `connection` | `Connection`

k_interval ::= `interval` | `Interval`

k_ident ::= `[a-zA-Z][a-zA-Z0-9_]*`

k_plus ::= `plus` | `Plus`

k_x ::= `x` | `X`

k_box ::= `box` | `Box`

k_diamond ::= `diamond` | `Diamond`

k_rectangle ::= `rectangle` | `Rectangle`

k_line ::= `line` | `Line`

k_where ::= `where` | `Where`

k_match ::= `match` | `Match`

k_lt ::= `<`

k_le ::= `<=`

k_eq ::= `==`

k_ge ::= >=

k_gt ::= >

k_ne ::= !=

k_and ::= &&

k_or ::= ||

k_not ::= !

k_number ::= [0-9]+

k_backwards ::= <-

k_forwards ::= ->

APPENDIX C. A NOTE ON EXPERIENCE USING THE FINALIZE FACILITY OF JAVA'S GARBAGE COLLECTION

This appendix discusses the experience we had using the Java Garbage Collector to automatically invoke the clean-up methods needed by the *Glimpse* Data Collection Facility.

The Java programming language provides a `finalize()` method that is invoked by the JVM for an object that is about to be garbage collected. The `finalize` mechanism allows an object to perform any clean up actions such as closing the file that it currently has open, before that object is freed by the system. We experimented with putting the code to properly close a log file inside a `finalize` method of a `ThreadData` object, thus eliminating the need to explicitly call the `closeThreadData()` and `close()` methods of the `GzEvent` object. One problem we encountered is due to the asynchronous nature of the Garbage Collector in the JVM. After a thread has exited its run method, the `finalize()` method might not be called until the execution of the program has continued on for quite some time. We wrote a short program to test the behavior of the Java Garbage Collector:

```
import java.util.*;
import java.io.*;
class invoke_final extends Thread{
    public invoke_final()
    {
        super();
        System.out.println("Create object: "+this.getName());
    }
    protected void finalize() throws Throwable {
        try {
           System.out.println("finalize method called on id "+
                              this.getName());
        } finally {
           super.finalize();
```

```
        }
    }
    public void run()
    {
        System.out.println("running id: "+this.getName());
    }
    public static void main(String[] args) {
        /* Forever loop that creates new threads. */
        while (true) {
            /* Create some threads and run them. */
            for (int i=0; i<100; i++) {
                invoke_final x=new invoke_final();
                x.start();
                Thread.yield();
            }
            /* Request JVM to run Garbage Collector */
            Runtime.getRuntime().gc();
            /* Create more threads and run them. */
            for (int i=51; i<100; i++) {
                invoke_final x=new invoke_final();
                x.start();
            }
        }
    }
}
```

In this test program, a forever `while` loop creates threaded objects. Each time through the `for` loop, an object is created and the thread associated with that object is started. At the beginning of each loop, the reference to the previously created object is removed. Since there is no longer any reference to the object in the main program, the JVM sees these unreachable objects as candidates for garbage collection. We inserted a call to the Java Garbage Collector in the middle of the while loop. When the Garbage Collector runs, it should invoke the `finalize()` method, which prints out a string indicating that the finalize method for this object has been invoked by the Garbage Collector.

When we ran this program, we saw different results under different JVM's. On the Windows NT system (SUN Java 1.1.8), at the point where we call

`Runtime.getRuntime().gc()`, we saw outputs indicating that the Garbage Collector ran. On the SGI (SUN Java 1.1.6), the Garbage Collector did not run until the outer `while` loop had gone through 10 to 20 iterations. Each iteration of the `while` loop creates 200 objects.

In summary, we would like the *Glimpse* data collection to close the log file when a given event thread exits, and not to wait until some time later. Unfortunately, the time of execution of the `finalized()` method, under the current JVM specification, is implementation-dependent. For example, placing the clean-up code inside the `finalize()` method works on the Windows NT system, but not on an SGI.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[1] Lantz Moore, Debra Hensgen, David Charley, Philip A. Wilsey, and Venkatram Krishnaswamy. Graze: A tool for performance visualization and analysis. *International Conference on Parallel Processing*, 1995.

[2] T. Hogan and T. Rosmond. The description of the navy operational global atmospheric prediction system's spectral forecast model. *Monthly Weather Review, 119*, 1991.

[3] R. Hodur. The naval research laboratory coupled ocean/atmosphere mesoscale prediction system (coamps). *Monthly Weather Review, 125*, 1997.

[4] Defense Modeling and Simulation Office. *Master Environmental Library (MEL), Technical Reference Guide, Characteristics and Performance, Version 1.1*, 1998.

[5] T. Hogan and T. Rosmond. The description of the navy operational global atmospheric prediction system's spectral forecast model. *Monthly Weather Review, 119*, 1991.

[6] Sharon E. Perl, William E. Weihl, and Brian Noble. *Continuous Monitoring and Performance Specification.* System Research Center, Report 153, 1998.

[7] Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer 28, 11, (November 1995), pages 37-46.*, 1995.

[8] Daniel A. Reed, Ruth A. Aydt, Luiz DeRose, Celso L. Mendes, Randy L. Ribler, Eric Shaffer, Hueseyin Scimitci, Jeffery S. Vetter, Daniel R. Wells, Shannon Whitmore, and Ying Zhang. *Performance Analysis of Parallel Systems: Approaches and Open Problems.* Department of Computer Science, University of Illinois, 1997.

[9] Randy L. Ribler, Jeffery S. Vetter, Huseyin Simitci, and Daniel A. Reed. Autopilot: Adaptive control of distributed applications. *Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, Chicago, IL, July 1998.*, 1998.

[10] I. Foster, C. Kesselman, and S. Tuecke. The nexus approach to integrating multithreading and communication. *Journal of Parallel and Distributed Computing, 37*, 1996.

[11] Randy L. Ribler, Huseyin Simitçi, and Daniel A. Reed. The autopilot peformance-directed adaptive control system. *Future Generation Computer Systems, special issue (Performance Data Mining), to appear.*, 1999.

[12] Daniel A. Reed, David A. Padua, Ian T. Foster, Dennis B. Gannon, and Barton P. Miller. Delphi: An integrated, language-directed performance prediction, measurement, and analysis environment. *The 9th Symposium on the Frontiers of Massively Parallel Computation, Annapolis, MD*, 1999.

[13] Dolphin Interconnect Solutions, Inc. *TimeScan, Event Analysis System User Guide, version 3.0*, 1997.

[14] Dolphin Interconnect Solutions, Inc. *TimeScan, ELOG Library Programmer's Guide, version 3.0.1*, 1997.

[15] Sharon E. Perl and William E. Weihl. Performance assertion checking. *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, 1993.

[16] Parallel Tools Consortium. *Manual Page for the Portable Timing Routines*, 1996.

[17] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification.* Addison-Wesley Pub Co., 1999.

[18] Silicon Graphics, Inc. *MIPSpro Fortran 77 Programmer's Guide*, 1999.

[19] Silicon Graphics, Inc. *UNIX MAN Page for MP, Multiprocessing directives for MIPS FORTRAN Compiler*, 1999.

[20] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley Longman, Inc., 1996.

[21] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification.* Addison-Wesley Longman, Inc., 1996.

[22] P. Brinch Hansen. A comparison of two synchronizing concepts. *Acta Informatica, Volume 1, Fasc. 3, (1972), pages 190-199.*, 1972.

[23] P. Brinch Hansen. Concurrent programming concepts. *Computing Surveys, Volume 5, Number 4, (December 1973), pages 223-245.*, 1973.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, CA 93943-5101

3. On-Scene Systems, Naval Research Laboratory, Monterey 1
   Code 7542
   7 Grace Hopper Ave.
   Monterey, CA 93943

4. Chairman, Computer Science 1
   Naval Postgraduate School
   Code CS,
   833 Dyer Rd.
   Monterey, CA 93943-5118

5. Debra Hensgen 1
   OpenTV, Inc.
   401 East Middlefield Road
   Mountain View, CA 94043

6. Rudy Darken 1
   Naval Postgraduate School
   Code CS/DR, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5118

7. Cynthia Irvine 1
   Naval Postgraduate School
   Code CS/IC, Computer Sciences Dept.
   833 Dyer Rd.
   Monterey, CA 93943-5118

8. Lantz Moore 1
   13249 Saddle Ridge Road
   Lakeside, CA 92040

9. Susie Tsai 1
   5534, South, Oakhurst Place
   Seattle, WA 98118

10. Pedro T.H. Tsai 1
    979 Syida Drive
    Pacific Grove, CA 93950